

Linux系统命令 及Shell脚本 实践指南

王军 著

Linux Commands and Shell Scripting

- 采用理论联系实际的方式，从系统管理出发，深入剖析Linux系统的运行原理，介绍Linux系统中的常用命令，理清Bash Shell编程的脉络。
- 结合作者多年的运维诊断经验，提供了大量实用性极强的脚本案例，对于广大Linux系统运维人员来说，可谓“一书在手，运维不愁”。



机械工业出版社
China Machine Press

Linux系统命令及Shell脚本实践指南

王军 著

ISBN: 978-7-111-44503-6

本书纸版由机械工业出版社于2014年出版，电子版由华章分社（北京华章图文信息有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @研发书局

腾讯微博 @yanfabook

目录

[推荐序1](#)

[推荐序2](#)

[前言](#)

[第1章 Linux简介](#)

[1.1 Linux的发展历史](#)

[1.2 Linux的特点](#)

[1.3 系统安装](#)

[1.3.1 安装前的规划](#)

[1.3.2 安装RedHat](#)

[1.3.3 安装CentOS](#)

[1.4 系统登录](#)

[1.4.1 第一次登录系统的设置](#)

[1.4.2 使用图形模式登录](#)

[1.4.3 使用终端模式登录](#)

[1.4.4 开始学习使用Linux的命令](#)

[1.5 系统启动流程](#)

[1.5.1 系统引导概述](#)

[1.5.2 系统运行级别](#)

[1.5.3 服务启动脚本](#)

[1.5.4 Grub介绍](#)

[1.6 获得帮助](#)

[1.6.1 使用man page](#)

[1.6.2 使用info page](#)

[1.6.3 其他获得帮助的方式](#)

[第2章 Linux用户管理](#)

[2.1 Linux用户和用户组](#)

[2.1.1 UID和GID](#)

[2.1.2 /etc/passwd和/etc/shadow](#)

[2.2 Linux账号管理](#)

[2.2.1 新增和删除用户](#)

[2.2.2 新增和删除用户组](#)

[2.2.3 检查用户信息](#)

[2.3 切换用户](#)

[2.3.1 切换成其他用户](#)

[2.3.2 用其他用户的身份执行命令：sudo](#)

[2.4 例行任务管理](#)

[2.4.1 单一时刻执行一次任务：at](#)

[2.4.2 周期性执行任务：cron](#)

[2.4.3 /etc/crontab的管理](#)

[第3章 Linux文件管理](#)

[3.1 文件和目录管理](#)

[3.1.1 绝对路径和相对路径](#)

[3.1.2 文件的相关操作](#)

[3.1.3 目录的相关操作](#)

[3.1.4 文件时间戳](#)

[3.2 文件和目录的权限](#)

[3.2.1 查看文件或目录的权限：ls-al](#)

[3.2.2 文件隐藏属性](#)

[3.2.3 改变文件权限：chmod](#)

[3.2.4 改变文件的拥有者：chown](#)

[3.2.5 改变文件的拥有组：chgrp](#)

[3.2.6 文件特殊属性：SUID/SGID/Sticky](#)

[3.2.7 默认权限和umask](#)

[3.2.8 查看文件类型：file](#)

[3.3 查找文件](#)

[3.3.1 一般查找：find](#)

[3.3.2 数据库查找：locate](#)

[3.3.3 查找执行文件：which/whereis](#)

[3.4 文件压缩和打包](#)

[3.4.1 gzip/gunzip](#)

[3.4.2 tar](#)

[3.4.3 bzip2](#)

[3.4.4 cpio](#)

[第4章 Linux文件系统](#)

[4.1 文件系统](#)

[4.1.1 什么是文件系统](#)

[4.1.2 ext2文件系统简介](#)

[4.1.3 ext3文件系统简介](#)

[4.2 磁盘分区、创建文件系统、挂载](#)

[4.2.1 创建文件系统：fdisk](#)

[4.2.2 磁盘挂载：mount](#)

[4.2.3 设置启动自动挂载：/etc/fstab](#)

[4.2.4 磁盘检验：fsck、badblocks](#)

[4.3 Linux逻辑卷](#)

[4.3.1 什么是逻辑卷](#)

[4.3.2 如何制作逻辑卷](#)

[4.4 硬链接和软链接](#)

[4.4.1 什么是硬链接](#)

[4.4.2 什么是软链接](#)

[第5章 字符处理](#)

[5.1 管道](#)

[5.2 使用grep搜索文本](#)

[5.3 使用sort排序](#)

[5.4 使用uniq删除重复内容](#)

[5.5 使用cut截取文本](#)

[5.6 使用tr做文本转换](#)

[5.7 使用paste做文本合并](#)

[5.8 使用split分割大文件](#)

[第6章 网络管理](#)

[6.1 网络接口配置](#)

[6.1.1 使用ifconfig检查和配置网卡](#)

[6.1.2 将IP配置信息写入配置文件](#)

[6.2 路由和网关设置](#)

[6.3 DNS客户端配置](#)

[6.3.1 /etc/hosts](#)

[6.3.2 /etc/resolv.conf](#)

[6.4 网络测试工具](#)

[6.4.1 ping](#)

[6.4.2 host](#)

[6.4.3 traceroute](#)

[6.4.4 常见网络故障排查](#)

[第7章 进程管理](#)

[7.1 什么是进程](#)

[7.2 进程和程序的区别](#)

[7.3 进程的观察：ps、top](#)

[7.4 进程的终止：kill、killall](#)

[7.5 查询进程打开的文件：lsof](#)

[7.6 进程优先级调整：nice、renice](#)

[第8章 Linux下的软件安装](#)

[8.1 源码包编译安装](#)

[8.1.1 编译、安装、打印HelloWorld程序](#)

[8.1.2 使用源码包编译安装Apache](#)

[8.2 RPM安装软件](#)

[8.2.1 什么是RPM](#)

[8.2.2 RPM包管理命令：rpm](#)

[8.2.3 包依赖关系](#)

[8.2.4 使用RPM包安装gcc](#)

[8.2.5 使用RPM包安装Apache](#)

[8.3 yum安装软件](#)

[8.3.1 yum命令的基本用法](#)

[8.3.2 使用yum安装Apache](#)

[8.3.3 RedHat使用yum的问题](#)

[8.3.4 自建本地yum源](#)

[8.3.5 自建网络yum源](#)

[8.4 三种安装方法的比较](#)

[8.5 重建RPM包](#)

- [8.5.1 创建重建环境](#)
- [8.5.2 快速重建RPM包](#)
- [8.5.3 以spec文件重建RPM包](#)
- [8.5.4 spec文件简介](#)

[第9章 vi和vim编辑器](#)

- [9.1 vi和vim编辑器简介](#)
- [9.2 vi编辑器](#)
 - [9.2.1 模式介绍](#)
 - [9.2.2 案例练习](#)
- [9.3 vim编辑器](#)
 - [9.3.1 多行编辑](#)
 - [9.3.2 多文件编辑](#)
 - [9.3.3 使用vimtutor深入学习vim](#)
- [9.4 gedit编辑器](#)
 - [9.4.1 gedit编辑器简介](#)
 - [9.4.2 启动gedit编辑器](#)

[第10章 正则表达式](#)

- [10.1 正则表达式基础](#)
 - [10.1.1 什么是正则表达式](#)
 - [10.1.2 基础的正则表达式](#)
 - [10.1.3 扩展的正则表达式](#)
 - [10.1.4 通配符](#)
- [10.2 正则表达式示例](#)
- [10.3 文本处理工具sed](#)
 - [10.3.1 sed介绍](#)
 - [10.3.2 删除](#)
 - [10.3.3 查找替换](#)
 - [10.3.4 字符转换](#)
 - [10.3.5 插入文本](#)
 - [10.3.6 读入文本](#)
 - [10.3.7 打印](#)
 - [10.3.8 写文件](#)

[10.3.9 sed脚本](#)

[10.3.10 高级替换](#)

[10.3.11 sed总结](#)

[10.4 文本处理工具awk](#)

[10.4.1 打印指定域](#)

[10.4.2 指定打印分隔符](#)

[10.4.3 内部变量NF](#)

[10.4.4 打印固定域](#)

[10.4.5 截取字符串](#)

[10.4.6 确定字符串的长度](#)

[10.4.7 使用awk求列和](#)

[第11章 Shell编程概述](#)

[11.1 Shell简介](#)

[11.1.1 Shell是什么](#)

[11.1.2 Shell的历史](#)

[11.1.3 Shell的功能](#)

[11.1.4 Shell编程的优势](#)

[11.2 第一个Shell脚本](#)

[11.2.1 编辑第一个Shell脚本](#)

[11.2.2 运行脚本](#)

[11.2.3 Shell脚本的排错](#)

[11.3 Shell的内建命令](#)

[第12章 Bash Shell的安装](#)

[12.1 确定你的Shell版本](#)

[12.2 安装bash](#)

[12.3 使用新版本的Bash Shell](#)

[12.4 在Windows中安装bash](#)

[第13章 Shell编程基础](#)

[13.1 变量](#)

[13.1.1 局部变量](#)

[13.1.2 环境变量](#)

[13.1.3 变量命名](#)

[13.1.4 变量赋值和取值](#)

[13.1.5 取消变量](#)

[13.1.6 特殊变量](#)

[13.1.7 数组](#)

[13.1.8 只读变量](#)

[13.1.9 变量的作用域](#)

[13.2 转义和引用](#)

[13.2.1 转义](#)

[13.2.2 引用](#)

[13.2.3 命令替换](#)

[13.3 运算符](#)

[13.3.1 算术运算符](#)

[13.3.2 位运算符](#)

[13.3.3 自增自减](#)

[13.4 其他算术运算](#)

[13.4.1 使用\\$\[\]做运算](#)

[13.4.2 使用expr做运算](#)

[13.4.3 内建运算命令declare](#)

[13.4.4 算术扩展](#)

[13.4.5 使用bc做运算](#)

[13.5 特殊字符](#)

[13.5.1 通配符](#)

[13.5.2 引号](#)

[13.5.3 注释符](#)

[13.5.4 大括号](#)

[13.5.5 控制字符](#)

[13.5.6 杂项](#)

[第14章 测试和判断](#)

[14.1 测试](#)

[14.1.1 测试结构](#)

[14.1.2 文件测试](#)

[14.1.3 字符串测试](#)

[14.1.4 整数比较](#)

[14.1.5 逻辑测试符和逻辑运算符](#)

[14.2 判断](#)

[14.2.1 if判断结构](#)

[14.2.2 if/else判断结构](#)

[14.2.3 if/elif/else判断结构](#)

[14.2.4 case判断结构](#)

[第15章 循环](#)

[15.1 for循环](#)

[15.1.1 带列表的for循环](#)

[15.1.2 不带列表的for循环](#)

[15.1.3 类C的for循环](#)

[15.1.4 for的无限循环](#)

[15.2 while循环](#)

[15.2.1 while循环的语法](#)

[15.2.2 使用while按行读取文件](#)

[15.2.3 while的无限循环](#)

[15.3 until循环](#)

[15.3.1 until循环的语法](#)

[15.3.2 until的无限循环](#)

[15.4 select循环](#)

[15.5 嵌套循环](#)

[15.6 循环控制](#)

[15.6.1 break语句](#)

[15.6.2 continue语句](#)

[第16章 函数](#)

[16.1 函数的基本知识](#)

[16.1.1 函数的定义和调用](#)

[16.1.2 函数的返回值](#)

[16.2 带参数的函数](#)

[16.2.1 位置参数](#)

[16.2.2 指定位置参数值](#)

[16.2.3 移动位置参数](#)

[16.3 函数库](#)

[16.3.1 自定义函数库](#)

[16.3.2 函数库/etc/init.d/functions简介](#)

[16.4 递归函数](#)

[第17章 重定向](#)

[17.1 重定向简介](#)

[17.1.1 重定向的基本概念](#)

[17.1.2 文件标识符和标准输入输出](#)

[17.2 I/O重定向](#)

[17.2.1 I/O重定向符号和用法](#)

[17.2.2 使用exec](#)

[17.2.3 Here Document](#)

[第18章 脚本范例](#)

[18.1 批量添加用户脚本](#)

[18.2 检测服务器存活](#)

[18.3 使用expect实现自动化输入](#)

[18.4 自动登录ftp备份](#)

[18.5 文件安全检测脚本](#)

[18.6 ssh自动登录备份](#)

[18.7 使用rsync备份](#)

[18.8 使用netcat备份](#)

[18.9 使用iptables建立防火墙](#)

[18.10 自定义开机启动项的init脚本](#)

[18.11 使用脚本操作MySQL数据库](#)

[18.12 基于LVM快照的MySQL数据库备份](#)

[18.13 页面自动化安装LAMP环境](#)

推荐序1

认识王军是在2008年，还记得当年他说是怀着对Linux及开源软件极大的兴趣来参加我的课程。作为一个Linux从业10年并在开源软件培训业从事教学多年的我而言，说“阅人无数”毫不夸张，也因此对很多学生都没有太深刻的印象。随着课程的开展和对王军逐渐的了解，我渐渐地感受到他好学的精神和他对技术的执着，也逐渐对其心生敬佩，并坚定地认为，只需假以时日他一定能在Linux领域有所成绩。

课程结束后至今，我和他一直保持着断断续续的联系，也经常能从其他朋友的口中听到他的消息，事实证明他的努力已经逐渐得到越来越多的认可。在此期间我也在不少技术论坛上看到他的技术帖，其中很多文字表现出他对技术独到的见解。

一个偶然的机会再次见到王军，并得知他正在写一本关于Linux系统管理和Shell编程方面的书，当时就希望能在初步成稿后一览为快。拿到初稿后，我花了几个小时仔细通读了一遍，可以肯定这确实是一本凝结了他多年实际工作经验的作品。与很多Linux书籍不同，该书尽量避免使用生硬的技术词汇，对知识点的描述都尽量地做到“去技术化”、“去概念化”，同时也能看出各章节前后的衔接顺序包括各个知识点的出场他都做了精心设计，并深入浅出地加以描述，这些都是我认为该书是一本很好的Linux系统和Shell编程入门书籍的原因，初学者完全可以通过该书迅速入门，但是要想完全吃透该书还需要读者结合实际工作多次研读。

最后希望王军的书能帮助和带领更多的Linux系统爱好者进入Linux的殿堂，同时也希望王军能再接再厉，再出好书！

上海尚观科技有限公司教学总监、技术总监、首席系统架构师 邹凯（Kissingwolf）

推荐序2

2010年有幸与王军共事，期间相处极为投缘。王军当时在公司主要负责推动平台虚拟化和自动化运维相关工作，在这段时间里王军给我的印象是为人热情，凡事有求必应；另一方面他非常热衷于技术研究，喜欢钻研新技术，喜欢突破创新，在那段时间里王军带着其他同事将公司原来的虚拟化平台进行了一次大规模升级，使公司系统更稳定，遗憾的是后来有了更好的发展机会他离开了51JOB。

得知王军的书稿已经完成并不日出版，为他高兴的同时也深感不易：他平时很忙且需要经常出差，能在日常繁忙工作的基础上，把一些知识要点记录下来已经是很少有人能做到的事情了，而且还能坚持整理成册，这绝对是需要毅力才能完成的工作。

借此机会分享一次与王军的聊天心得，这也是我们Linux服务器运维工程师心灵成长的点滴记录。运维工作已经不是搬搬服务器、扛扛交换机、配配网络的时代了，现在运维工作应该以“降低成本，提升用户体验”为目标。降低成本就无形要求运维技能的提升，如现在流行的“去IOE”，去掉高端、昂贵小型机服务器就必须用多台廉价的PC服务器代替，但又要保证系统稳定、高可用、可扩展性强，这样就要求运维工程师具备过硬的Linux技能。在提升用户体验的过程中，有三点极其重要：一是稳定，不能频繁宕机；二是要快，天下武功，唯快不破；三是界面友好，不能半天找不到操作按钮——这些都是与精湛的技术密不可分的。

收到王军的定稿并邀请写序，突然感觉到压力，一是自己很少写序，怕写不好；二是担心影响了王军的努力成果。在仔细阅读后，感觉本书最大特点是结构清晰、各章节前后贯穿、

知识框架完整且覆盖全面，伴有大量深入浅出的案例，无疑是一本很好的Linux系统运维和Shell编程的作品，我很钦佩作者的技术功底。

本书从基础知识入手，系统讲解了Linux系统结构、主流服务器搭建及故障排除、用户权限管理、磁盘存储管理、文件系统管理、内存管理、进程管理、Shell编程等关键技术，同时，王军根据他多年的运维诊断经验，提供了大量实用性极强的脚本案例，对于广大Linux服务器运维人员来说，可谓“一书在手，运维不愁”。

51JOB系统经理兼架构师 杨向勇

前言

为什么要写这本书

早在我还在大学校园时就对Linux产生了极大的兴趣，期间我热衷于下载、安装、体验各种不同的Linux发行版，并尝试使用Linux作为我的桌面系统。但实际情况是，由于大学中使用群体极小，学校又没有开设直接的Linux系统课程，虽然我花了不少的课余时间去研究它，但始终感觉不得其法，难以入门。至今我依然记得当时使用鼠标双击好不容易才复制桌面上的rpm包，并抱怨为什么没有出现类似于Windows的“安装向导”。所以实际上有很长一段时间，面对Linux系统我能做的少之又少。

2006年大学毕业后，我有幸进入了一直梦寐以求的IT行业，并从此正式走上了技术之路。工作中能实际接触到Linux系统运维是我在该领域发展的很重要的外部因素，工作的驱使和个人的兴趣成为我每天坚持学习Linux的源动力。但当时很尴尬的一个现实是：一方面互联网行业的高速发展促进了Linux如火如荼的发展，另一方面又很难找到真正适合“新手”的入门级教材，得到一本简单明了的入门书籍是我当时迫切的愿望。于是在走了不少弯路并感觉自己已经“迷路”之后，我报名参加了Linux系统工程师社会培训班，利用工作之余系统并完整地学习了Linux。事实证明，当时的选择是正确的，这直接影响了我的职业发展乃至今后的职业规划。

经历了多年的工作后，我也非常希望能有机会与大家分享自己在IT领域的体会，所以也经常在一些技术网站发表技术文章，或是与志同道合的朋友一起举办免费的网络培训班。但是总体来说，所涉及的内容大多零碎、不成体系。筹划本书的初期，我想把重点放在Linux系统管理、高性能计算、高可用集

群甚至云计算这些“够时髦”的主题上，但是反复思考后觉得，“时髦”的技术永远在变，而且限制了读者范围。但是对我、对很多梦想学习Linux的读者来说这更是一个机会：用最简单、最朴素、最基础的语言讲解和描述Linux系统以及如何使用它，给更多初学者以“可以学会”的希望和“努力前行”的力量。

出于这样的考虑，我在组织本书的内容时尽量安排书的各个章节以及每章节中的每个小节，甚至是每小节中的知识点的出现顺序符合新手的认知规律，做到从易到难，从基础到提高，以循序渐进的方式将各类知识点以人物出场、层次推进的方式呈现在读者面前，尽量避免将生僻的术语突然摆在读者面前，造成读者思维上的困扰，并且尽量使用简单明了的文字和浅显易懂的比喻帮助读者理解、消化。尽管如此，我还是希望读者能在此基础上展开阅读，并根据实际需要做必要的深入理解。

总之，这是一本讲解Linux系统和Shell编程的入门级书籍，内容主要涉及Linux的基础命令、编辑器的使用和Shell脚本的开发。

读者对象

本书适合以下读者阅读：

- Linux爱好者
- Linux初学者
- 希望学习Shell编程的读者
- 希望了解系统的网络工程师
- 网站前后台开发人员

如何阅读本书

本书从知识结构上分为三大部分，第一部分（第1～8章）为基础内容，详细地介绍了Linux的历史发展、安装使用、用户管理、文件管理、文件系统、字符处理、网络管理、进程管理和软件安装。第二部分（第9～10章）为编辑器部分，内容为Linux下常用编辑器vi和vim的用法和基于流处理的sed和awk工具，这是管理Linux系统的基本技能。第三部分（第11～18章）为Shell编程，内容包括Shell的安装、使用、语法，其中最后一章是本部分的重点，该章所有脚本在实际应用中有很高的使用率。

如果你是Linux的初学者，我建议从第1章开始阅读。第二部分的内容涉及vi和vim编辑器的操作细节，建议读者通读。如果读者已经有一定的基础，希望学习Shell脚本开发，可以直接跳至第三部分学习。

勘误和支持

由于作者的水平有限，编写的时间也很仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。如果你有更多的宝贵意见，欢迎你发送邮件至我的邮箱 johnwang.wangjun@gmail.com，或是关注我的新浪微博 weibo.com/u/1186347743，我很期待能够听到你们的真挚反馈。

致谢

首先，感谢伟大的Linux之父Linus Torvalds，他最初开发的这套Linux系统已经改变了整个世界的面貌，也影响了我个人的职业乃至人生发展。

感谢那些无数个为了解决问题而彻夜无眠的夜晚，感谢那

些在寒冬凌晨的三四点接到报警电话后集体上阵的兄弟姐妹们，感谢那些年一起扛起24×7运维任务的战友们，这一切无疑是我人生中最宝贵的财富。

感谢机械工业出版社华章公司的编辑杨绣国（Lisa），感谢她在这一年多的时间里始终支持我的写作，她的鼓励和帮助引导我顺利完成全部书稿。

谨以此书，献给我最亲爱的家人，以及众多热爱Linux的朋友们。

王军

2013年10月于上海

第1章 Linux简介

1.1 Linux的发展历史

首先我们一起来了解一下应该怎么读Linux这个单词，根据Torvalds（Linux的发明者）在其多次公开场合中的说明，标准的读音应该是“哩呐科斯”，利用搜索引擎加关键字Linux pronunciation进行搜索，就可以看到具体的视频。

说到Linux就不得不提到UNIX，因为Linux是一种类UNIX的系统。早在1965年，贝尔实验室加入了一项由美国通用电气公司和麻省理工学院发起的合作计划，该计划要开发一个多用户、多进程、多层次的Multics操作系统。由于该计划实际进展太过缓慢，1969年便暂停了。不过该计划的参与者之一Ken Thompson已经从这项计划中获得了一些点子和收获，当时他有一个被称为“星际旅行”的程序在GE-635的机器上运行，因为该机器性能问题，运行“星际旅行”太慢，从而引发了他想将这个程序移植到一台性能更好的DPD-7上的想法，只是因为家中有小孩需要照顾而一直没有时间动手。巧合的是，在1969年8月左右，他的妻儿出门探亲了一个月，就在这一个月的时间里，Thompson编写了一个操作系统，并成功地将“星际旅行”移植到了DPD-7上，而这个操作系统就是UNIX的原型。

UNIX由于具有优秀的移植性而得到了广泛的关注和支持，1974年12月伯克利大学获得UNIX的源码，并动手将其修改为适合自己机器的版本，最终命名为BSD，这也是UNIX很重要的一个分支。由于当时还没有足够的版权意识，很多商业公司都开始了基于UNIX操作系统的开发，比如AT&T的System V、IBM的AIX等，在这段时期中也形成了UNIX的两大分支：System V和BSD。

后来AT&T公司出于商业考虑（贝尔实验室是从属于AT&T公司的），1979年在发行第七版UNIX时开始严格限制对学生提供源码。这对大学教学影响非常大，因为在无法看到源码的情况下，教学工作便很难进行。当时有个叫Tanenbaum的教授为避免版权纠纷，在完全不看UNIX源码的情况下，自己动手写了一个类UNIX的系统，并命名为Minix，这项工作从1984年持续到1986年。由于开发这个系统的出发点在于教学，所以用户对Minix的新需求往往得不到开发支持，只能基于Minix的源码自己进行修改。

1984年，Richard Stallman创立了GNU项目，由自由软件基金支持，GNU项目的目标是“开发一个完全自由的UNIX操作系统”。

“Hello everybody out there using minix,I'm doing a free operation system”，1991年8月，网络上出现了以此开篇的帖子，这是一名芬兰的大学生为了写一个类Minix的系统而在寻找志同道合的伙伴，他就是著名的Linux之父——Linus Torvalds。同年10月5日，他在网络上发布了大约有1万行代码的Linux 0.01版本，次年已经有约1000人在使用Linux了。1993年，大约有100名程序员参与了Linux内核开发工作，其中核心人员有5名，此时Linux 0.99版本的代码大约有10万行，用户约为10万人。1994年，Linux加入了GNU，成为GNU项目中的一员，同年Linux 1.0版本发布，代码量大约有17万行，最早按照完全自由免费的协议发布，用户可以随意下载、使用、修改，而不需要通知作者。随后采用了GPL协议，很多开发人员开始将自己的代码贡献给核心小组，这也就使得当时的Linux系统对不同硬件都有着极好的支持，大大提高了不同平台间的可移植性。1995年，Linux可以在Intel、Digital等主流处理器上运行，用户量超过50万。1996年，Linux 2.0版本发布，并支持多处理器，此时的Linux进入实用阶段，用户量已经达到350万。1998年，RedHat公司宣布商业支持计划，迅猛推进了Linux的

发展，至此Linux正式成为真正的服务器操作系统并继续成长。

1.2 Linux的特点

从1991年问世到今天，Linux在服务器、桌面、行业定制等各级领域都获得了长足的发展，尤其在服务器领域获得了令人瞩目的成就，被业界认为是未来最有前途的操作系统之一。在嵌入式领域，由于Linux具有良好的移植性、丰富的代码资源等优点，也受到了越来越多的关注。下面我们就来看看这个操作系统有哪些主要特点。

第一，免费开源。Linux是一款完全免费的操作系统，任何人都可以从网络上下载到它的源代码，并可以根据自己的需求进行定制化的开发，而且没有版权限制。

第二，模块化程度高。Linux的内核设计分成进程管理、内存管理、进程间通信、虚拟文件系统、网络5部分，其采用的模块机制使得用户可以根据实际需要，在内核中插入或移走模块，这使得内核可以被高度的剪裁定制，以方便在不同的场景下使用。

第三，广泛的硬件支持。得益于其免费开源的特点，有大批程序员不断地向Linux社区提供代码，使得Linux有着异常丰富的设备驱动资源，对主流硬件的支持极好，而且几乎能运行在所有流行的处理器上。

第四，安全稳定。Linux采取了很多安全技术措施，包括读写权限控制、带保护的子系统、审计跟踪、核心授权等，这为网络环境中的用户提供了安全保障。实际上有很多运行Linux的服务器可以持续运行长达数年而无须重启，依然可以性能良好地提供服务，其安全性已经在各个领域得到了广泛的证实。

第五，多用户，多任务。多用户是指系统资源可以同时被

不同的用户使用，每个用户对自己的资源有特定的权限，互不影响。多任务是现代化计算机的主要特点，指的是计算机能同时运行多个程序，且程序之间彼此独立，Linux内核负责调度每个进程，使之平等地访问处理器。由于CPU处理速度极快，从用户的角度来看所有的进程好像在并行运行。

第六，良好的可移植性。Linux中95%以上的代码都是用C语言编写的，由于C语言是一种机器无关的高级语言，是可移植的，因此Linux系统也是可移植的。

1.3 系统安装

1.3.1 安装前的规划

可能会有读者正计划学习Linux而苦恼于不知道使用哪一个发行版，其实所有的发行版不管是RedHat、CentOS还是Ubuntu，其内核都是来自Linux内核官网（www.kernel.org），不同发行版之间的差别在于软件管理的不同，所以不管使用哪一个发行版，只要理解其原理之后，各类发行版的区别其实不大。当然对于初学者来说，拥有广泛的学习资源也是很重要的。由于RedHat公司进行了大力商业推广，且得益于其成熟的认证体系，因此使用RedHat的用户比较多，同时，它还有丰富的相关技术文档，以及活跃的社区，所以作为入门学习，可以使用RedHat。不过，近年来，CentOS发展也很迅猛，这个发行版和RedHat几乎完全一样，而且在某些方面还比RedHat略胜一筹，所以在本书中后面的所有内容中将主要使用版本为5.5的CentOS，小部分涉及RedHat的内容也将采用5.5版本。

有读者可能会考虑在一台计算机上安装多个操作系统，比如说在自己的家用计算机上安装Windows用于娱乐和日常应用或Windows环境下的开发等，另外再安装Linux系统用于学习。在这种情况下，最简单的安装方法是先安装Windows，后安装Linux，这样开机的时候就自动出现操作系统选择条，可以根据实际需要选择进入不同的操作系统。

由于Linux对系统的需求并不高，所以几乎所有计算机都可以安装，但是考虑到入门学习Linux需要用到图形界面，所以建议最好不要低于以下配置：CPU，P-3 800MHz；内存，1GB；硬盘，40GB。

在安装Linux的过程中，必须要有的两个分区为根分区

(/)和swap分区（交换分区），当然还有一些其他的分区可以独立出来，比如说/boot分区、/var分区等。

另外，这里介绍几个概念，便于大家理解后面即将出现的一些专业词汇。

什么是交换分区？交换分区是一个特殊的分区，它的作用相当于Windows下的虚拟内存，这个分区的大小一般设置为物理内存的两倍，但是不管物理内存有多大，交换分区建议不要超过8GB，因为大于8GB的交换分区其实并没有多大实际意义。

什么是Grub？Grub是一个系统引导工具，通过它可以加载内核，从而引导系统启动。

什么是/boot分区？/boot分区用于放置Linux启动所用到的文件，如kernel和initrd文件。

什么是DHCP？DHCP是Dynamic Host Configuration Protocol的简写，中文称为动态主机配置协议。在TCP/IP网络中，每台主机都需要有IP地址才能与其他主机通信，在一个大规模的网络中，如果由管理员手动地对每一台主机进行IP地址配置是不现实的。由此也就产生了DHCP协议，可用它来对网络节点上的主机进行IP地址配置。

1.3.2 安装RedHat

本节将演示安装RedHat系统的过程，使用到的版本是RedHat 5.5。大家可以先到网上下载RedHat 5.5操作系统的ISO文件，然后刻成光盘再安装。当然不要忘记在计算机的主板中设置从光驱启动，也可以使用虚拟机软件通过安装虚拟机的方式模拟安装过程。

计算机从光盘启动后，首先会显示如图1-1所示界面（注意看英文提示）。如果想使用图形界面安装直接按回车键即可，或者在10秒之内不做任何输入，这样也会默认进入图形安装模式。如果想用字符模式安装，需要输入`linux text`，然后按回车键。如果计算机的内存过小，安装程序会检测到因内存不足而无法进入图形安装模式，转而进入字符安装模式。



图1-1 光盘启动界面

这里选择使用图形模式安装，所以直接按回车键。接下来会针对硬件进行一些检测，并加载一些基本的驱动，然后就到了欢迎界面，如图1-2所示。

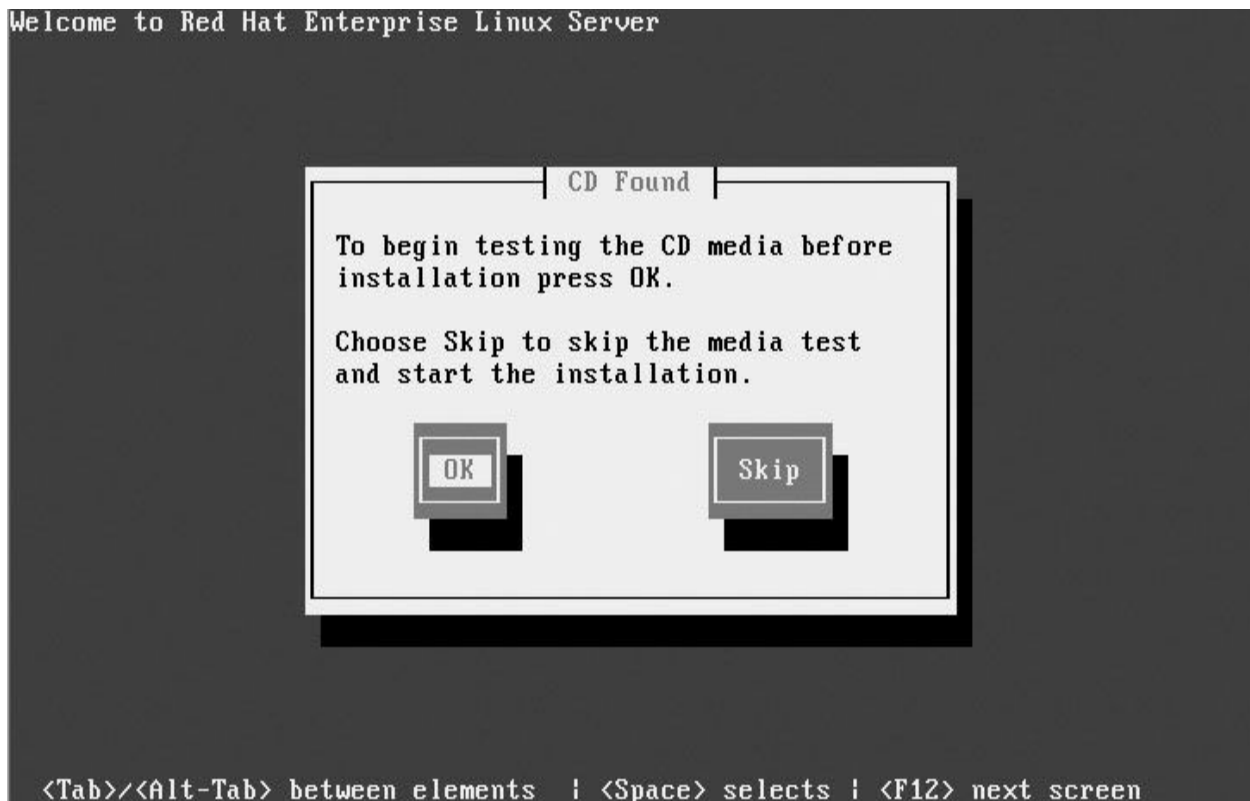


图1-2 介质检查界面

这里提供了安装介质的检测功能，一般来说只要下载后的ISO文件所使用的MD5比对值和官方给出的值一样，就说明安装介质没有问题，直接略过即可。略过方法是按Tab键使光标跳至Skip按钮，然后按回车键，这时会载入一个叫做anaconda的安装程序，如图1-3所示。它会调出图形安装界面。

A terminal window with a dark background and light-colored text. The text shows the progress of the anaconda installer. The first line says 'Running anaconda, the Red Hat Enterprise Linux Server system installer - please wait...'. The second line says 'Probing for video card: VMware SVGA II Adapter'.

```
Running anaconda, the Red Hat Enterprise Linux Server system installer - please
wait...
Probing for video card:  VMware SVGA II Adapter
```

图1-3 加载anaconda安装程序

注意看图1-3中的文字：Running anaconda,the Red Hat Enterprise Linux Server system installer，这句话说明anaconda其实是RedHat系统的安装工具。

成功加载了图形安装界面后，单击Next按钮进入下一步，如图1-4所示。



图1-4 anaconda启动的图形界面

接下来要选择安装过程中使用的语言，默认选择 English（English），单击Next按钮进入下一步，如图1-5所示。

在选择计算机使用的键盘时，使用默认U.S.English，单击Next按钮进入下一步，如图1-6所示。

进入如图1-7所示的界面后，会提示输入安装序列号。只有

在购买了RedHat的官方服务后，才能得到这个序列号。这里读者可能会有疑问：RedHat不是免费的吗，怎么会有序列号呢？RedHat确实是免费使用的，但是RedHat同时也提供了一些收费服务，购买了这些收费的服务后，RedHat官方将会给予相应的技术支持，这就是需要序列号的原因。这里直接略过，选择Skip entering Installation Number，然后单击OK按钮。




图1-5 安装过程中的语言选择


RED HAT ENTERPRISE LINUX 5



Select the appropriate keyboard for the system.

- Slovenian
- Spanish
- Swedish
- Swiss French
- Swiss French (latin1)
- Swiss German
- Swiss German (latin1)
- Tamil (Inscript)
- Tamil (Typewriter)
- Turkish
- U.S. English**
- U.S. International
- Ukrainian
- United Kingdom

 [Release Notes](#)

 [Back](#)


 [Next](#)

图1-6 键盘类型选择

RED HAT ENTERPRISE LINUX 5



Select the appropriate keyboard for the system.

Slovenian
Spanish
Swedish
Swiss French
Swiss French (latin1)
Swiss German
Swiss German (latin1)
Tamil (Inscript)
Tamil (Typewriter)
Turkish
U.S. English
U.S. International
Ukrainian
United Kingdom

Installation Number

Would you like to enter an Installation Number (sometimes called Subscription Number) now? This feature enables the installer to access any extra components included with your subscription. If you skip this step, additional components can be installed manually later.

See <http://www.redhat.com/InstNum/> for more information.

☒ Installation Number:

☐ Skip entering Installation Number

Back

OK

Release Notes

Back

Next

图1-7 输入安装序列号

这时会弹出一个确认窗口，再次单击Skip按钮，如图1-8所示。



图1-8 确认窗口

安装过程其实就是将系统装入磁盘，所以这里会弹出一个警告，提示是否初始化磁盘，这个操作会清除磁盘上的所有数据，单击Yes按钮，如图1-9所示。如果是在实际生产环境中安装，请一定要注意提前备份数据。



图1-9 确认初始化磁盘

接下来到了提示分区的页面。单击下拉框，然后选择 Create custom layout，单击Next按钮进入下一步，如图1-10所示。

RED HAT ENTERPRISE LINUX 5

Installation requires partitioning of your hard drive.
By default, a partitioning layout is chosen which is
reasonable for most users. You can either choose
to use this or create your own.

Remove all partitions on selected drives and create default layout.

Remove linux partitions on selected drives and create default layout.

Use free space on selected drives and create default layout.


Create custom layout.

Select the drive(s) to use for this installation.

☒ sda 20473 MB VMware, VMware Virtual S

+ Advanced storage configuration

☐ Review and modify partitioning layout

 Release Notes

 Back

 Next

图1-10 选择分区方式

在如图1-11所示的界面中可以创建分区，单击New按钮创建分区。

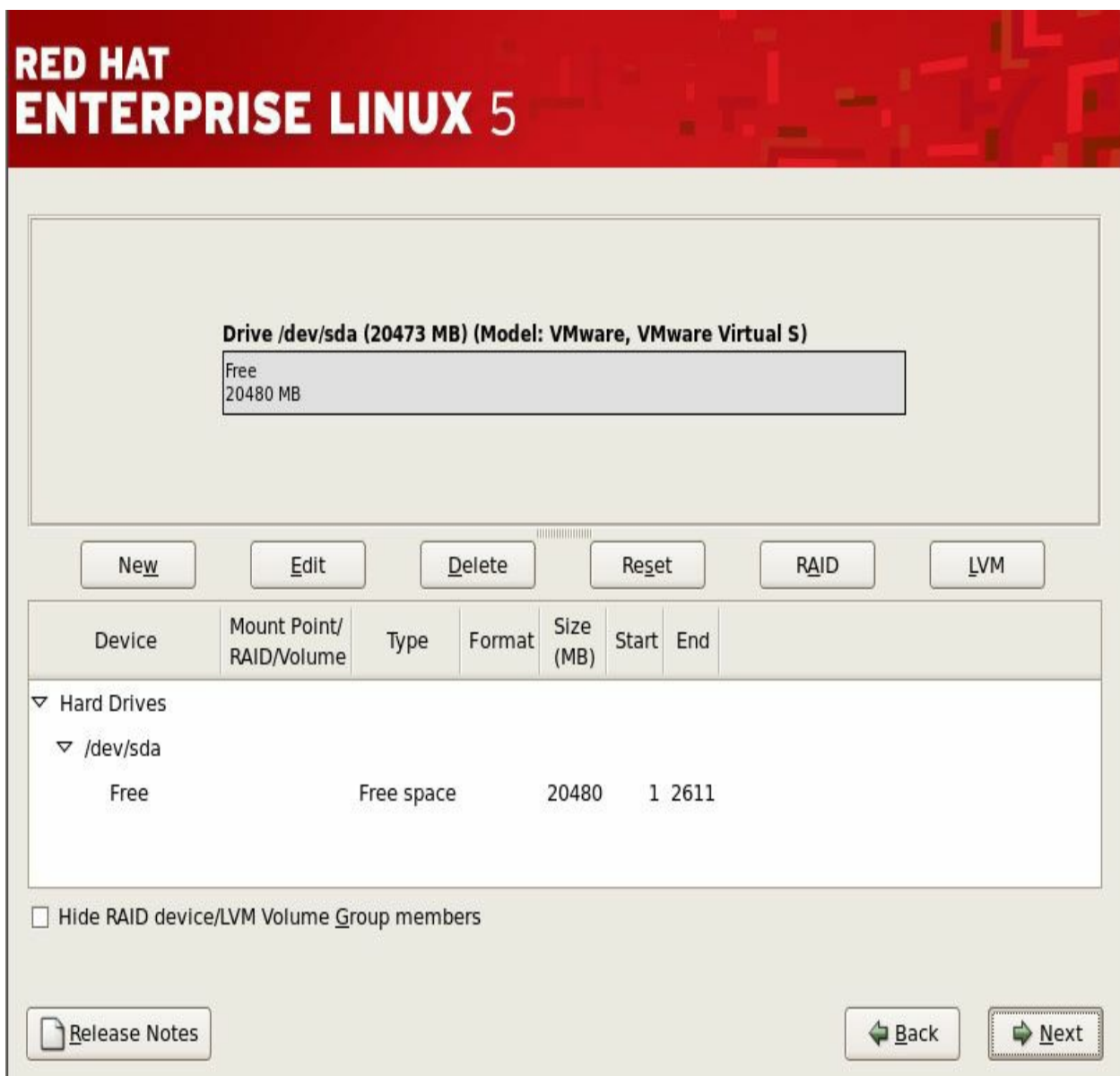


图1-11 创建磁盘分区

在如图1-12所示的界面中，Mount Point选择/boot，File System Type选择ext3，Size输入200。设置好后，单击OK按钮，然后再次单击New按钮创建第二个分区。

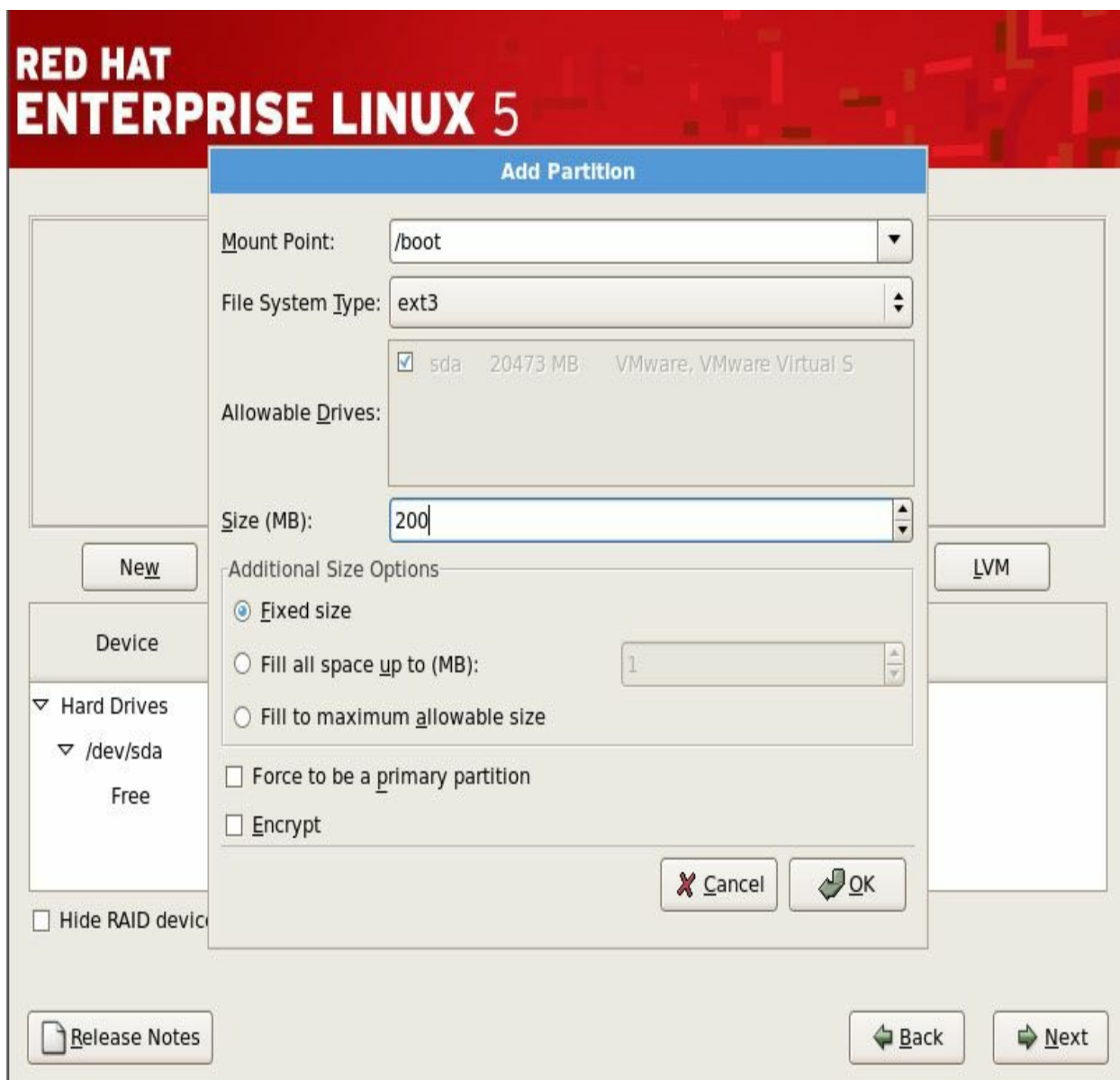


图1-12 创建/boot分区

swap分区是安装Linux系统必备的分区，按照之前对swap分区大小的说明，笔者使用的机器的内存为1024MB，所以这里设置为2048MB，如图1-13所示。单击OK按钮后再次单击New按钮创建第三个分区。

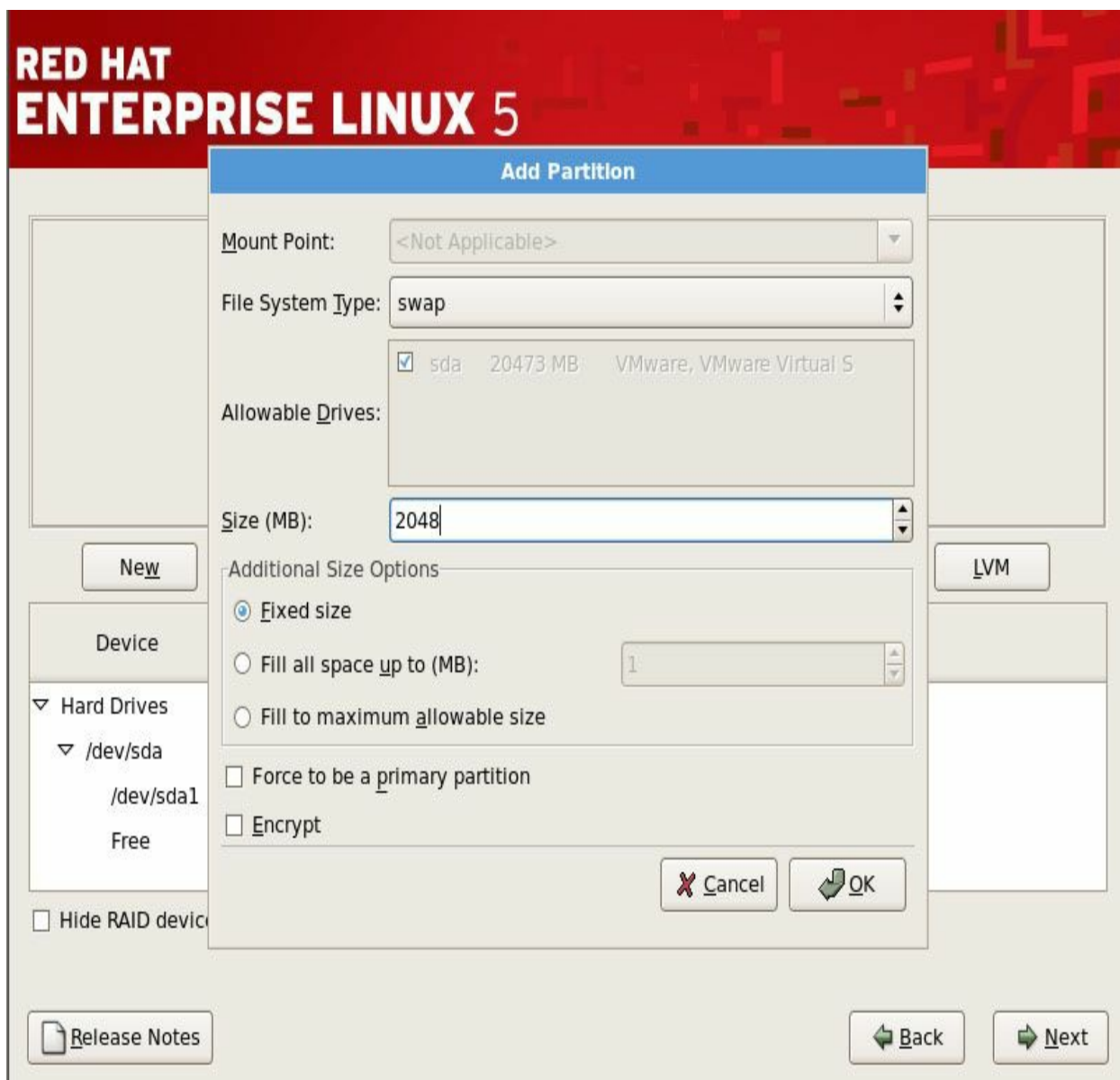


图1-13 创建swap分区

在如图1-14所示的界面中，把其他所有可用的空间都划为根分区（/），Mount Point选择“/”，File System Type选择ext3，在Additional Size Options中选择Fill to maximum allowable size。然后单击OK按钮，确认分区没有问题后，单击Next按钮进入下一步。

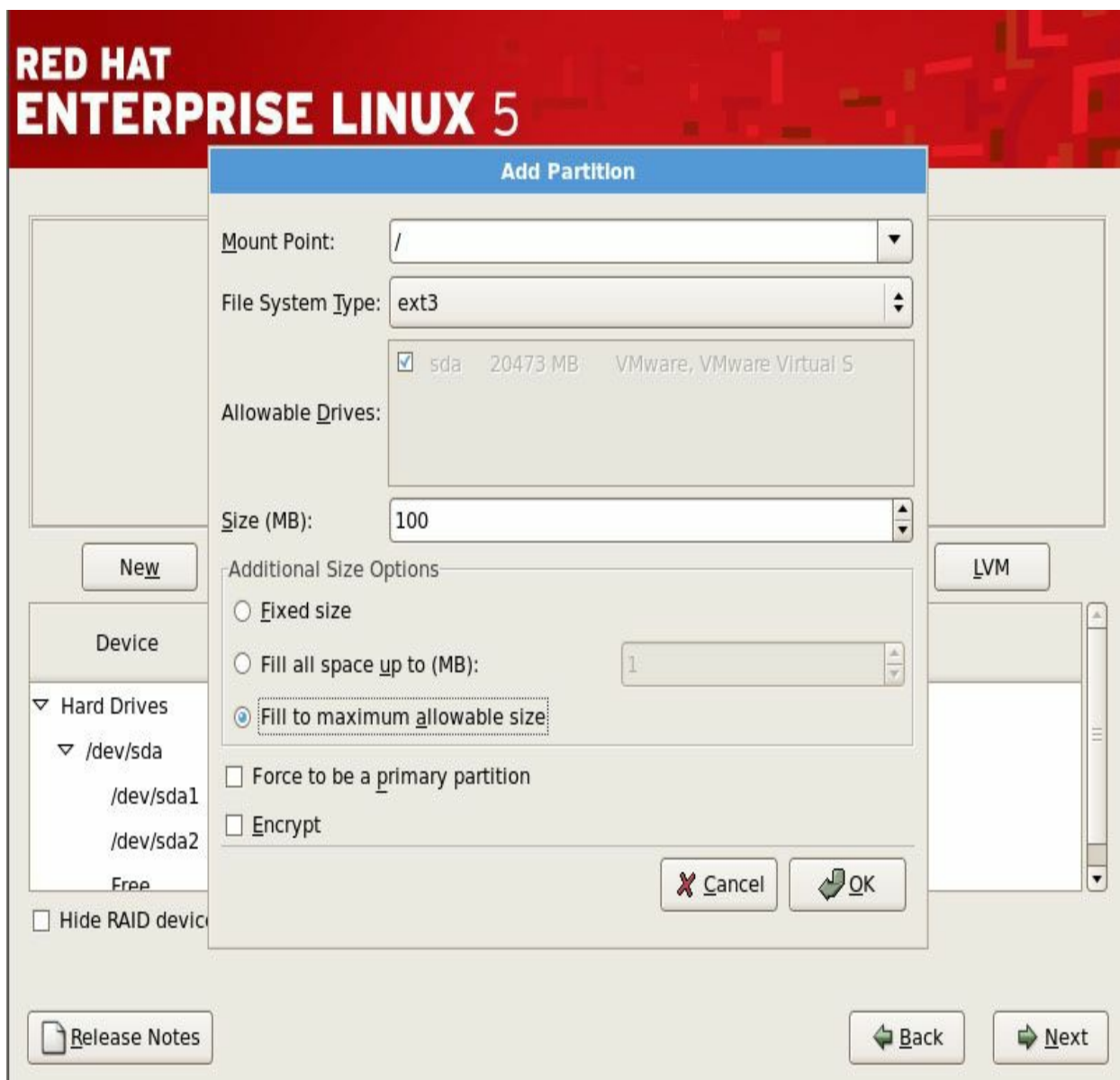


图1-14 创建根分区

到了安装Grub的部分，使用默认的设置即可，单击Next按钮进入下一步，如图1-15所示。

RED HAT ENTERPRISE LINUX 5

- ☒ The GRUB boot loader will be installed on /dev/sda.
- ☐ No boot loader will be installed.

You can configure the boot loader to boot other operating systems. It will allow you to select an operating system to boot from the list. To add additional operating systems, which are not automatically detected, click 'Add.' To change the operating system booted by default, select 'Default' by the desired operating system.

Default	Label	Device	
<input checked="" type="checkbox"/>	Red Hat Enterprise Linux Server	/dev/sda3	<div>Add</div> <div>Edit</div> <div>Delete</div>

A boot loader password prevents users from changing options passed to the kernel. For greater system security, it is recommended that you set a password.

☐ Use a boot loader password

Change password

☐ Configure advanced boot loader options

Release Notes

Back

Next

图1-15 安装Grub

图1-16是网卡配置，使用默认的配置，即自动从DHCP获得地址，单击Next按钮进入下一步。如果读者采用的是物理主机安装，请确保服务器网络环境中DHCP服务器，如果没有，需要单击manually手工设置IP地址。

RED HAT ENTERPRISE LINUX 5

Network Devices

Active on Boot	Device	IPv4/Netmask	IPv6/Prefix	Edit
<input checked="" type="checkbox"/>	eth0	DHCP	Auto	

Hostname

Set the hostname:

☒ automatically via DHCP


☐ manually (e.g., host.domain.com)

Miscellaneous Settings

Gateway:

Primary DNS:

Secondary DNS:

 [Release Notes](#)

 [Back](#)

 [Next](#)

图1-16 网卡配置

设置时区时，选择Asia/Shanghai，然后单击Next按钮进入下一步。有个快捷的办法，使用鼠标在地图上单击中国上海的位置，就可以迅速地设置好时区，如图1-17所示。



图1-17 时区选择

设置root密码时，输入两次同样的密码后，单击Next按钮进入下一步，如图1-18所示。为了安全起见，建议使用包含数字、大小写字母、特殊字符，长度至少为6位的密码。

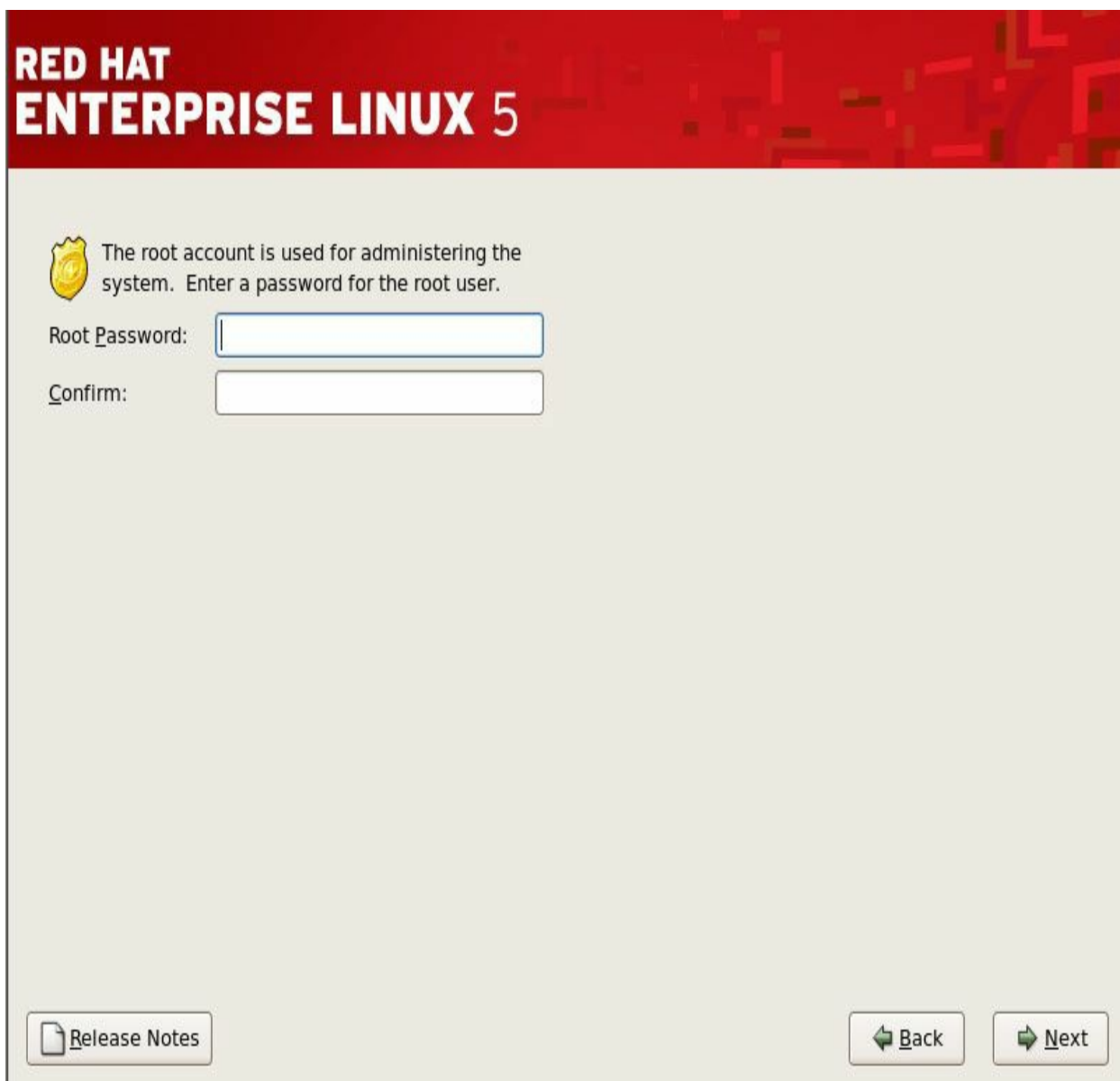


图1-18 设置root密码

在图1-19所示的界面中可以对预装的包做一些选择，如果单击Customize now，然后单击Next按钮，就会进入预装包的选择页面。因为我们需要的包可以后期再安装，所以这里直接使用默认选项，单击Next按钮进入下一步。


RED HAT ENTERPRISE LINUX 5


The default installation of Red Hat Enterprise Linux Server includes a set of software applicable for general internet usage. What additional tasks would you like your system to include support for?

- ☐ Software Development
- ☐ Web server

You can further customize the software selection now, or after install via the software management application.

☒ Customize later ☐ Customize now

 [Release Notes](#)

 [Back](#)


 [Next](#)

图1-19 定制包界面

这时安装程序会进行安装包的依赖关系的判定，然后跳至如图1-20所示的最终安装界面。


RED HAT ENTERPRISE LINUX 5



Click next to begin
installation of Red Hat
Enterprise Linux Server.

A complete log of the
installation can be found in
the file `'/root/install.log'`
after rebooting your system.

A kickstart file containing
the installation options
selected can be found in the
file `'/root/anaconda-ks.cfg'`
after rebooting the system.

 [Release Notes](#)

 [Back](#)

 [Next](#)

图1-20 系统安装确认

如果这时候想起来有什么需要修改的话，可以单击Back按钮后退修改配置，如果确认一切设置正确，就可以单击Next按钮，之后便开始格式化分区，并进入真正的安装过程了，如图1-21所示。



图1-21 正式安装过程

正式安装系统时，视系统配置不同，安装过程可能会持续几分钟到十几分钟不等，这里需要做的只是耐心等待。


安装结束后，需要重启以进入刚刚安装的系统，单击 Reboot按钮，如图1-22所示。至此RedHat系统的安装就结束了。


RED HAT ENTERPRISE LINUX 5



Congratulations, the installation is complete.

Remove any media used during the installation process and press the "Reboot" button to reboot your system.

 [Release Notes](#)

 [Back](#)


 [Reboot](#)

图1-22 安装结束

1.3.3 安装CentOS

CentOS与RedHat的安装过程大同小异，本节将演示CentOS的完整安装过程。本例中所采用的版本与之前安装的RedHat一致，即5.5版本。当计算机从光盘启动后，首先将会显示如图1-23所示的启动界面。

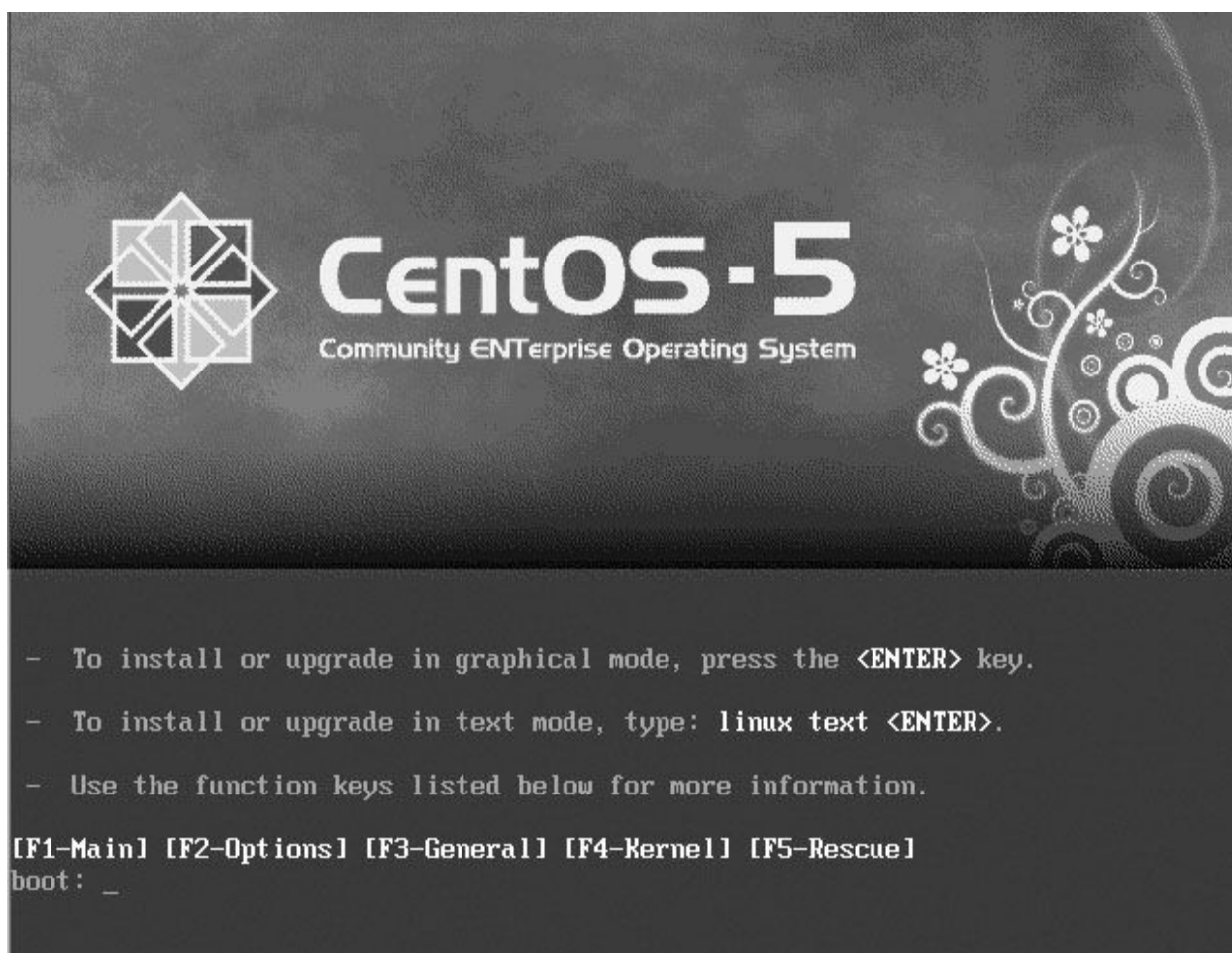


图1-23 光盘引导界面

同样，在这里直接按回车键将进入图形安装模式，如果计算机检测到内存太小，将会自动进入字符安装模式；或者输入“linux text”，按回车键后进入字符安装模式。这里直接按回车键开始安装过程。

安装介质检测时，按Tab键使光标跳至Skip按钮，按回车键确认，如图1-24所示。



图1-24 介质检查界面

开始运行anaconda，调出图形安装界面，如图1-25所示。

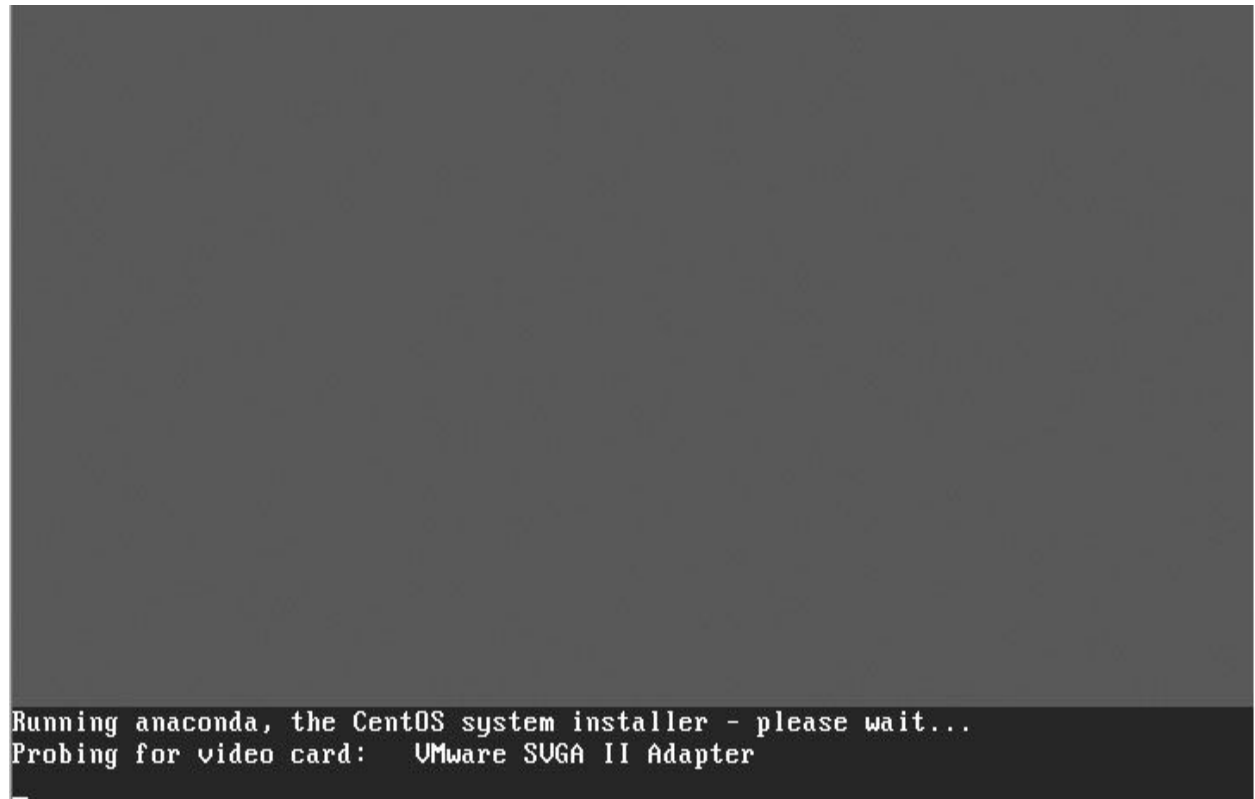


图1-25 加载anaconda安装程序

图形界面成功启动，直接单击Next按钮进入下一步，如图1-26所示。



图1-26 anaconda启动的图形界面

选择安装过程中使用的语言，默认选择 English（English），单击Next按钮进入下一步，如图1-27所示。



图1-27 安装过程中的语言选择

选择计算机使用的键盘时，使用默认的U.S.English，单击Next按钮进入下一步，如图1-28所示。

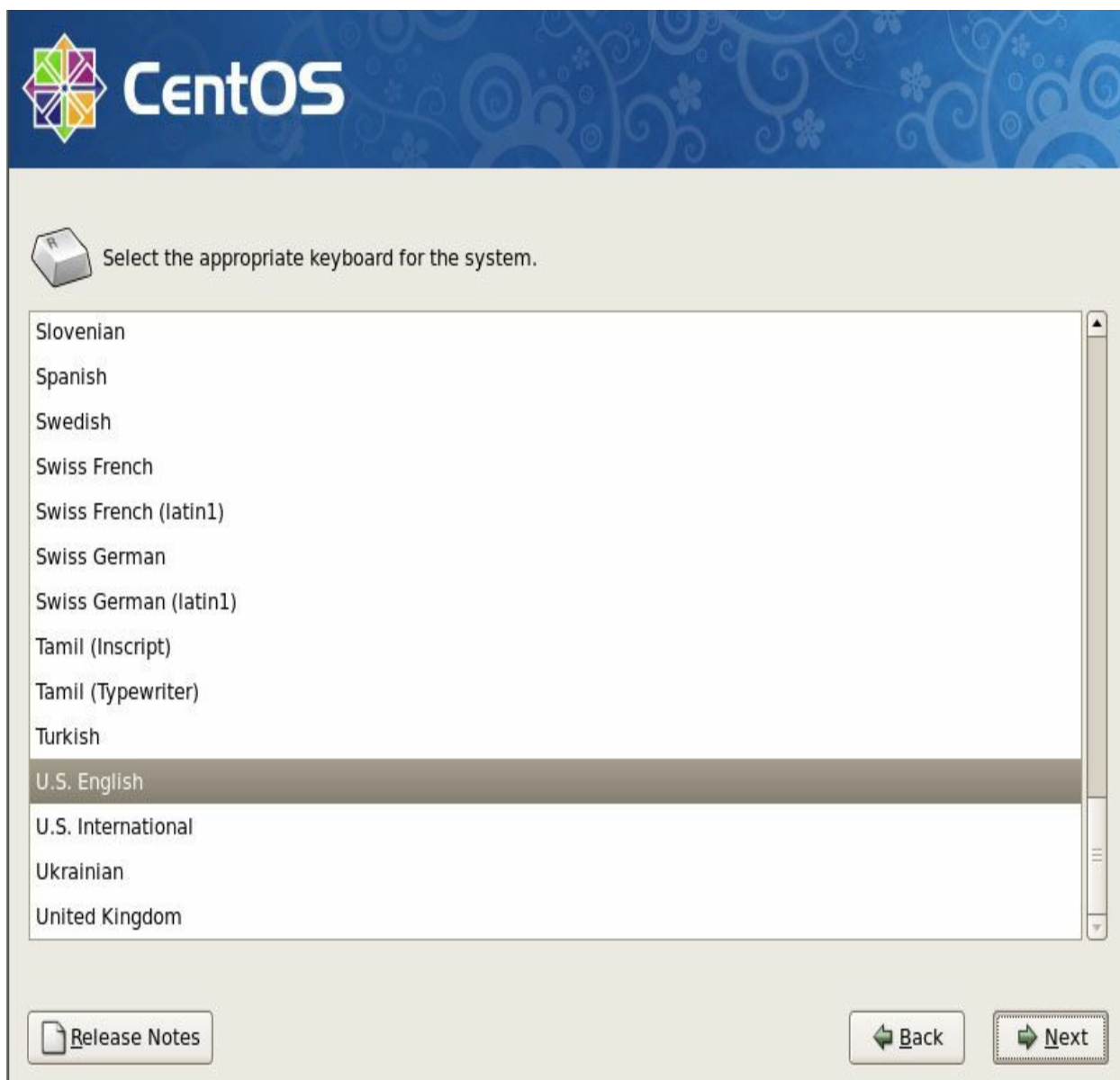


图1-28 键盘类型选择

接下来会提示安装过程中将会初始化磁盘并删除数据，如果在生产环境中安装系统，请确认之前已经做好备份。单击Yes按钮进入下一步，如图1-29所示。

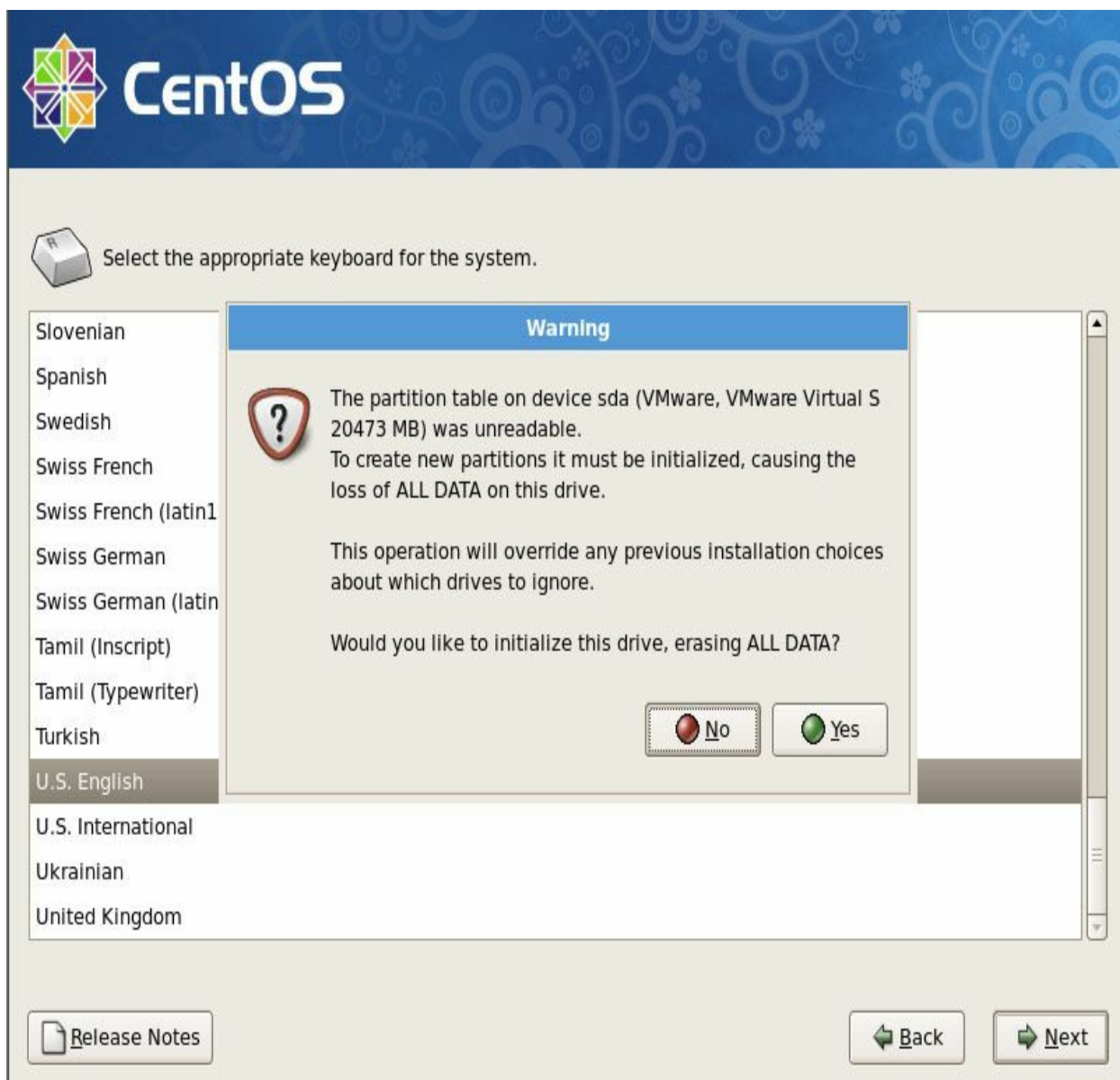


图1-29 确认初始化磁盘

进入分区设置后，单击下拉框选择Create custom layout，然后单击Next按钮，如图1-30所示。

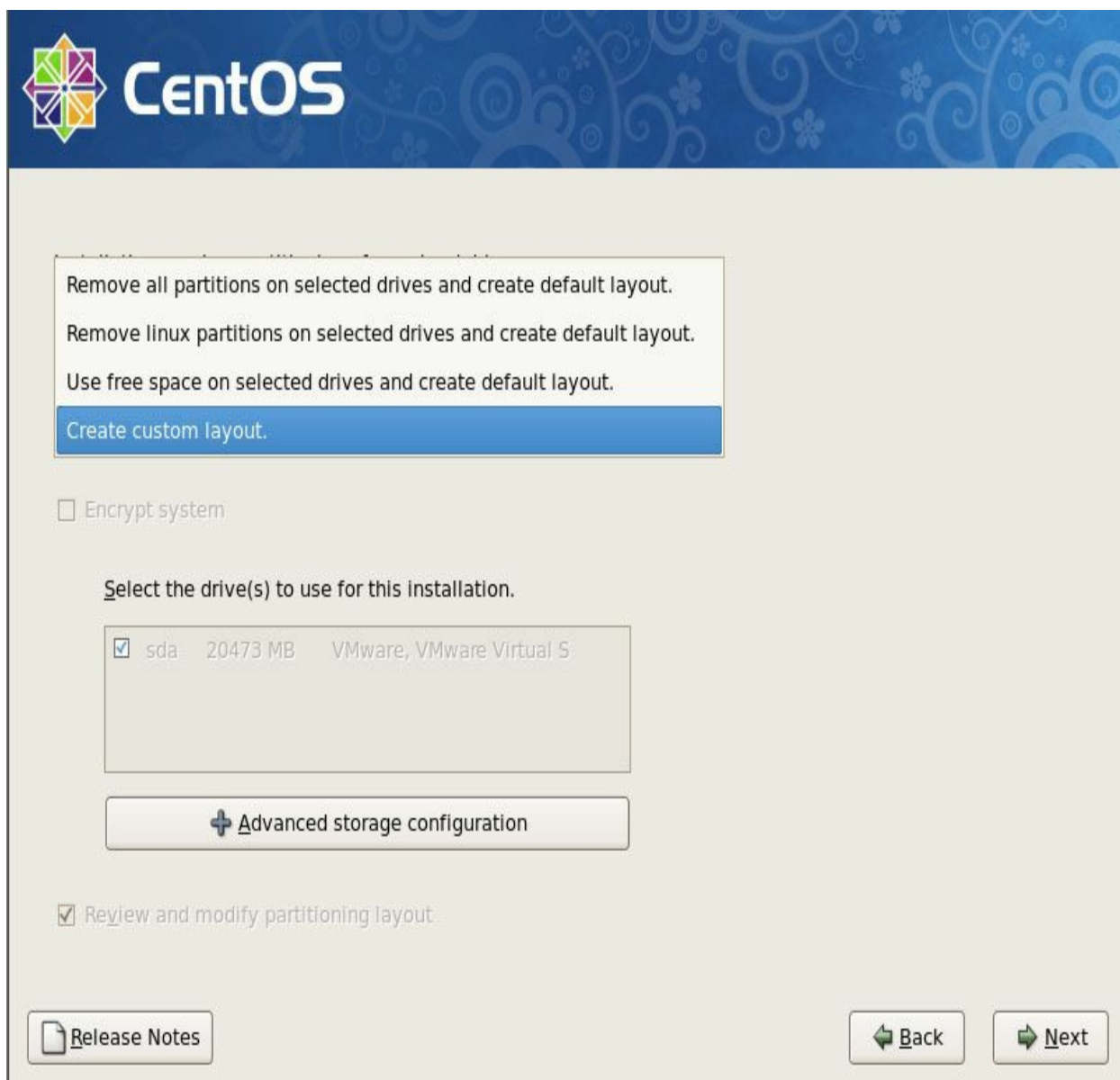


图1-30 选择分区方式

在图1-31所示的界面中开始创建分区，单击New按钮创建一个新的分区。

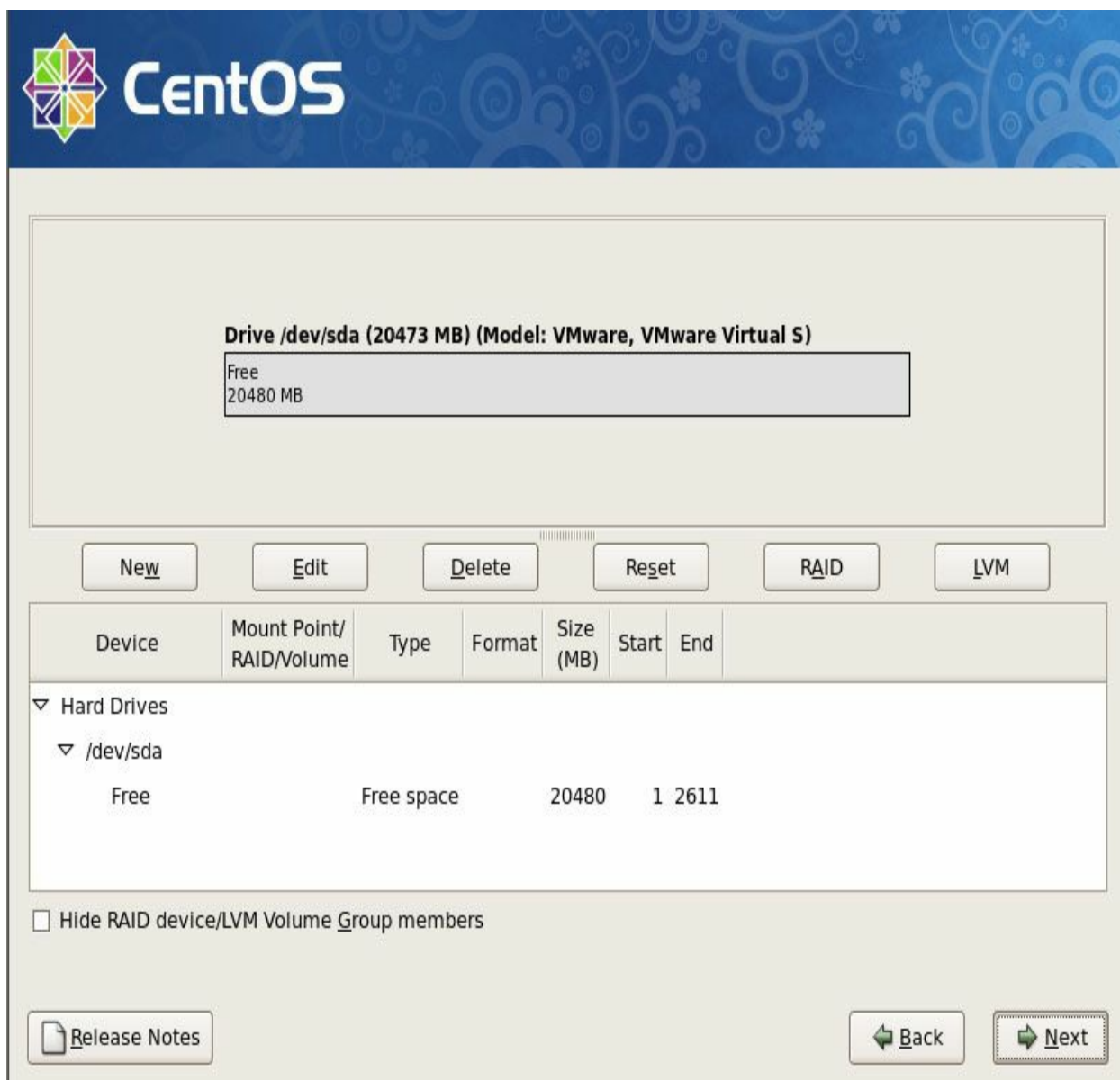


图1-31 创建分区

与之前安装RedHat分区的方式一样，选择200MB的/boot分区，2048MB的swap分区，其他所有可用空间分配给根分区，具体分区方式如图1-32所示。确认分区无误后，单击Next按钮进入下一步。

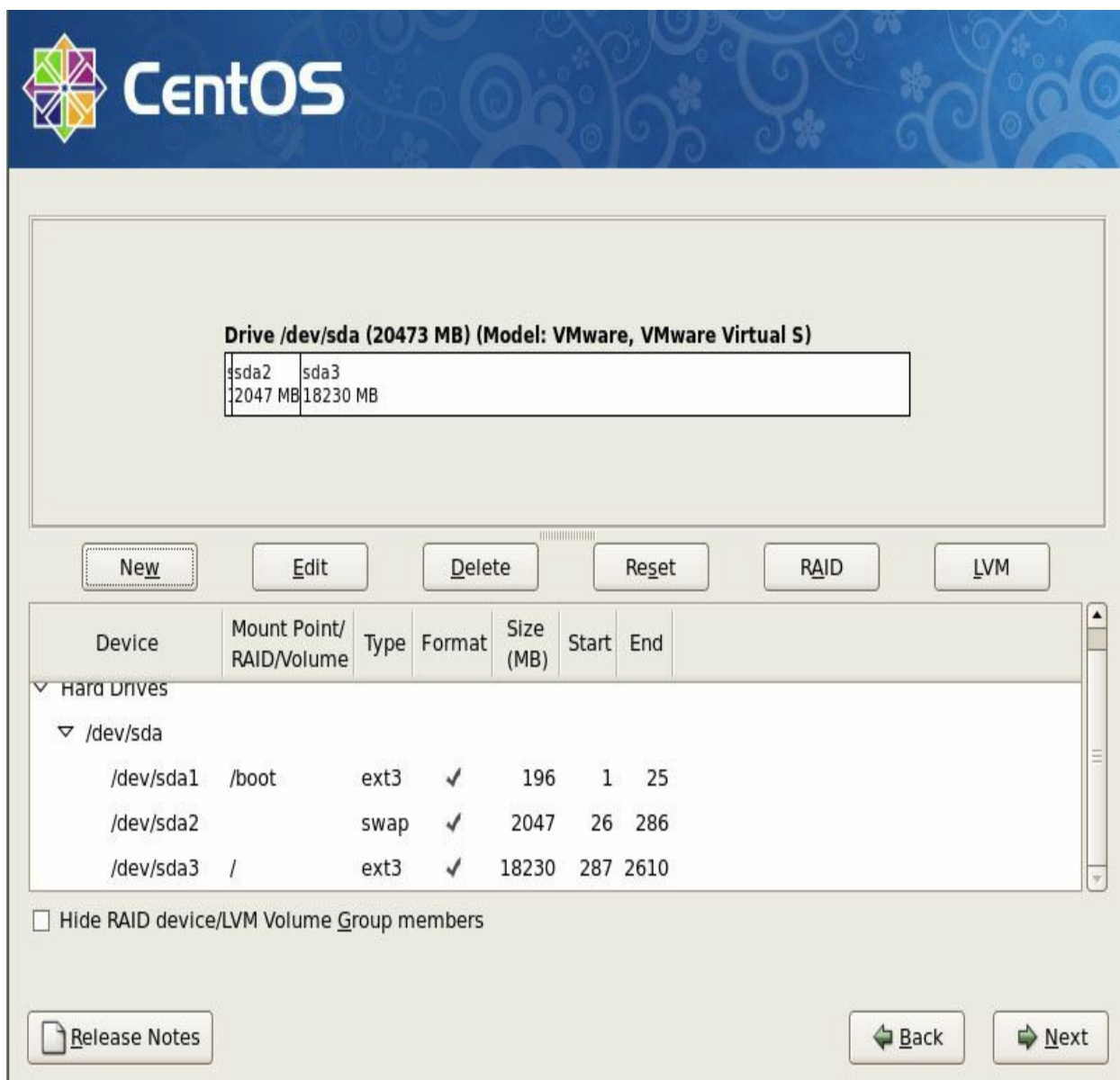


图1-32 最终分区显示

在Grub配置界面，使用默认配置，直接单击Next按钮，如图1-33所示。

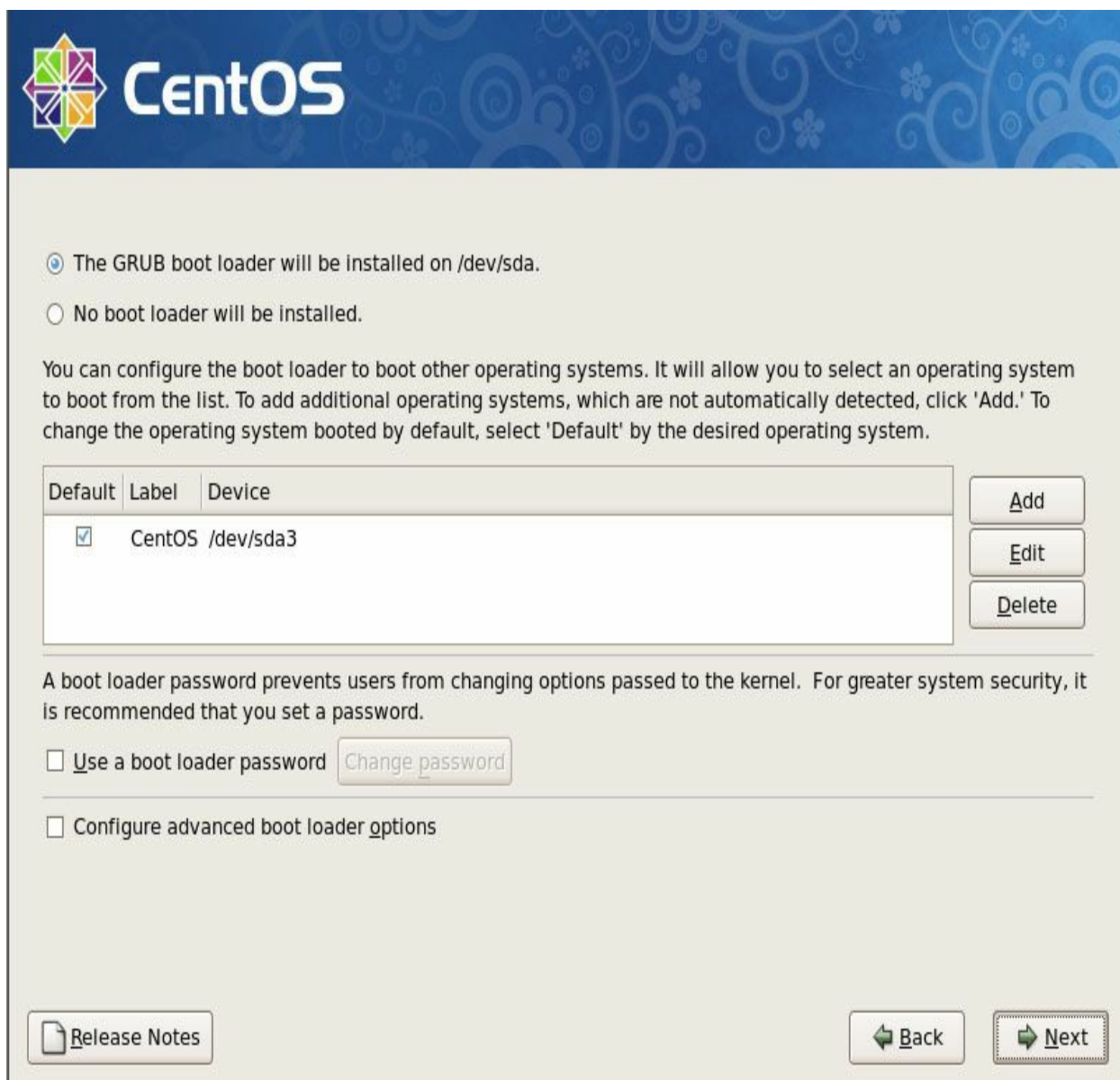


图1-33 安装Grub

进入网卡配置界面后，使用默认的DHCP获得网络配置，单击Next按钮进入下一步，如图1-34所示。



CentOS

Network Devices

Active on Boot	Device	IPv4/Netmask	IPv6/Prefix
<input checked="" type="checkbox"/>	eth0	DHCP	Auto
<div></div>			

Edit

Hostname

Set the hostname:

☒ automatically via DHCP

☐ manually (e.g., host.domain.com)

Miscellaneous Settings

Gateway:

Primary DNS:

Secondary DNS:

Release Notes

Back

Next

图1-34 网卡配置界面

时区的设置选择Asia/Shanghai，然后单击Next按钮，如图1-35所示。

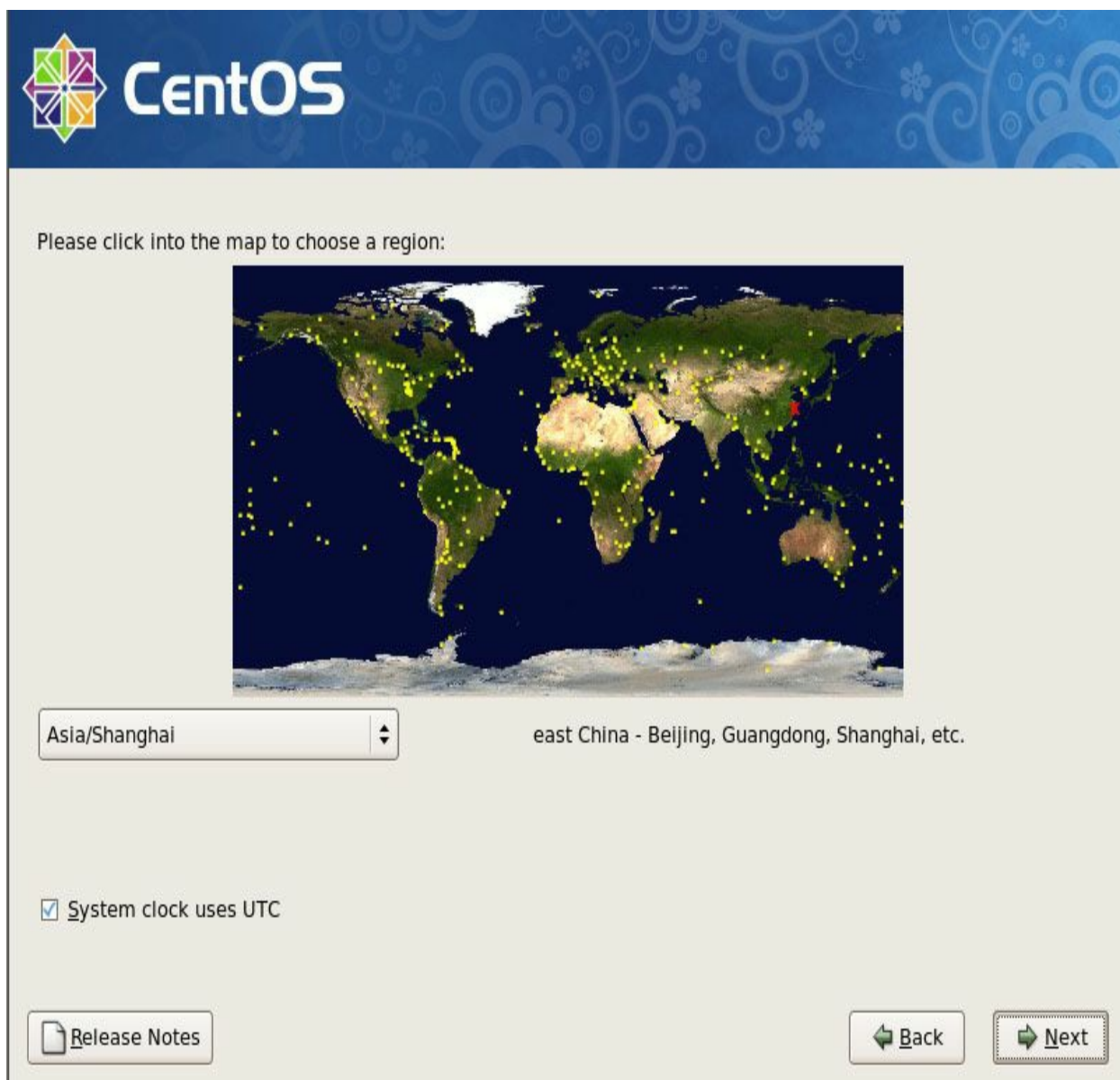


图1-35 时区设置

设置root密码时，两次输入一样的密码后，单击Next按钮，如图1-36所示。



图1-36 设置root密码

接下来选择预装包，如果选择Customize now，然后单击Next按钮，就可以立即对预装的包做选择。这里采用默认值，直接单击Next按钮即可，如图1-37所示。

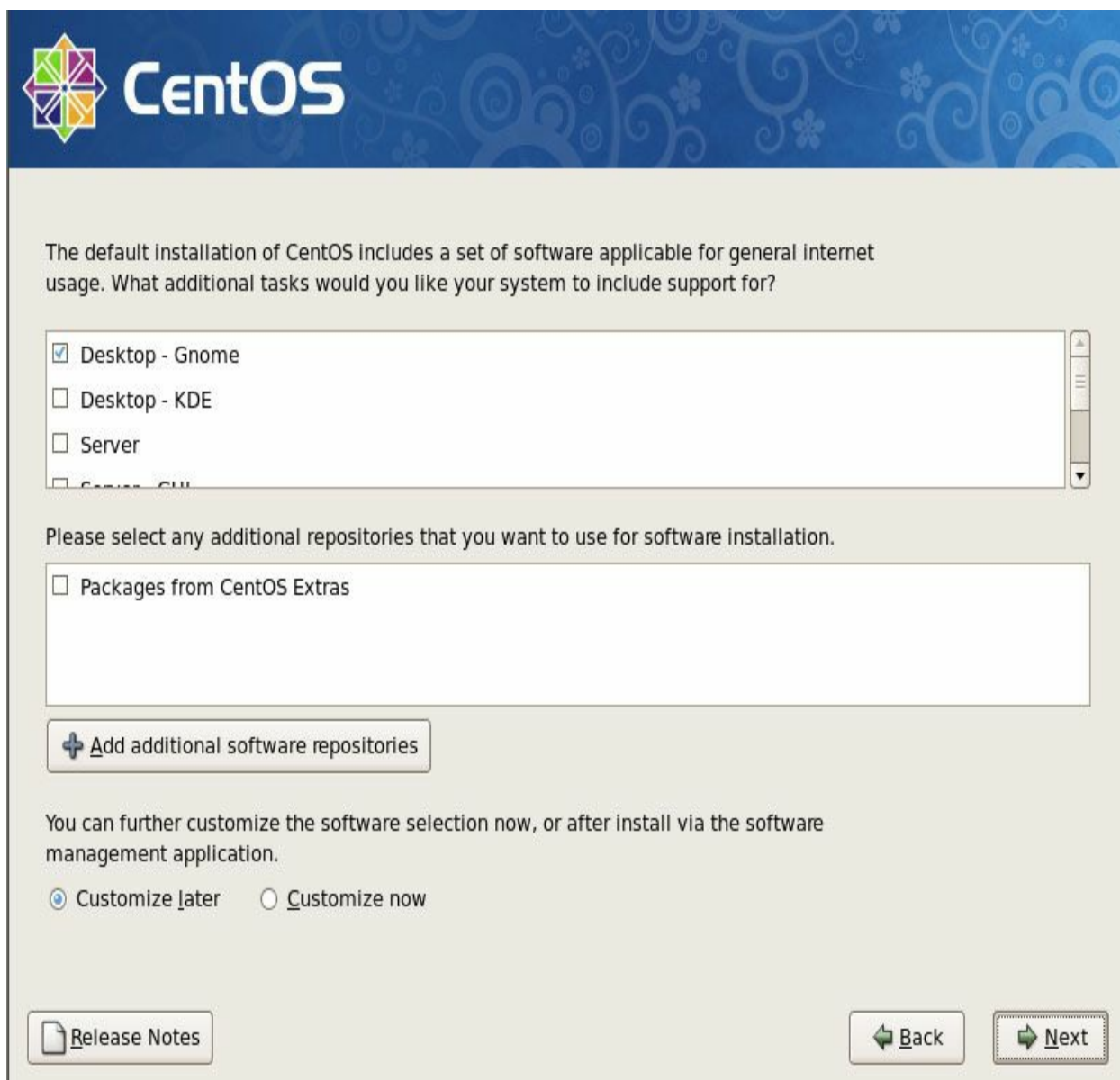


图1-37 包定制界面

在如图1-38所示的界面中单击Next按钮进入实际的安装过程。首先格式化分区、检查安装中的包依赖关系，然后开始安装系统。视计算机性能不同，安装过程可能持续几分钟到十几分钟不等，如图1-39所示。



图1-38 系统安装确认

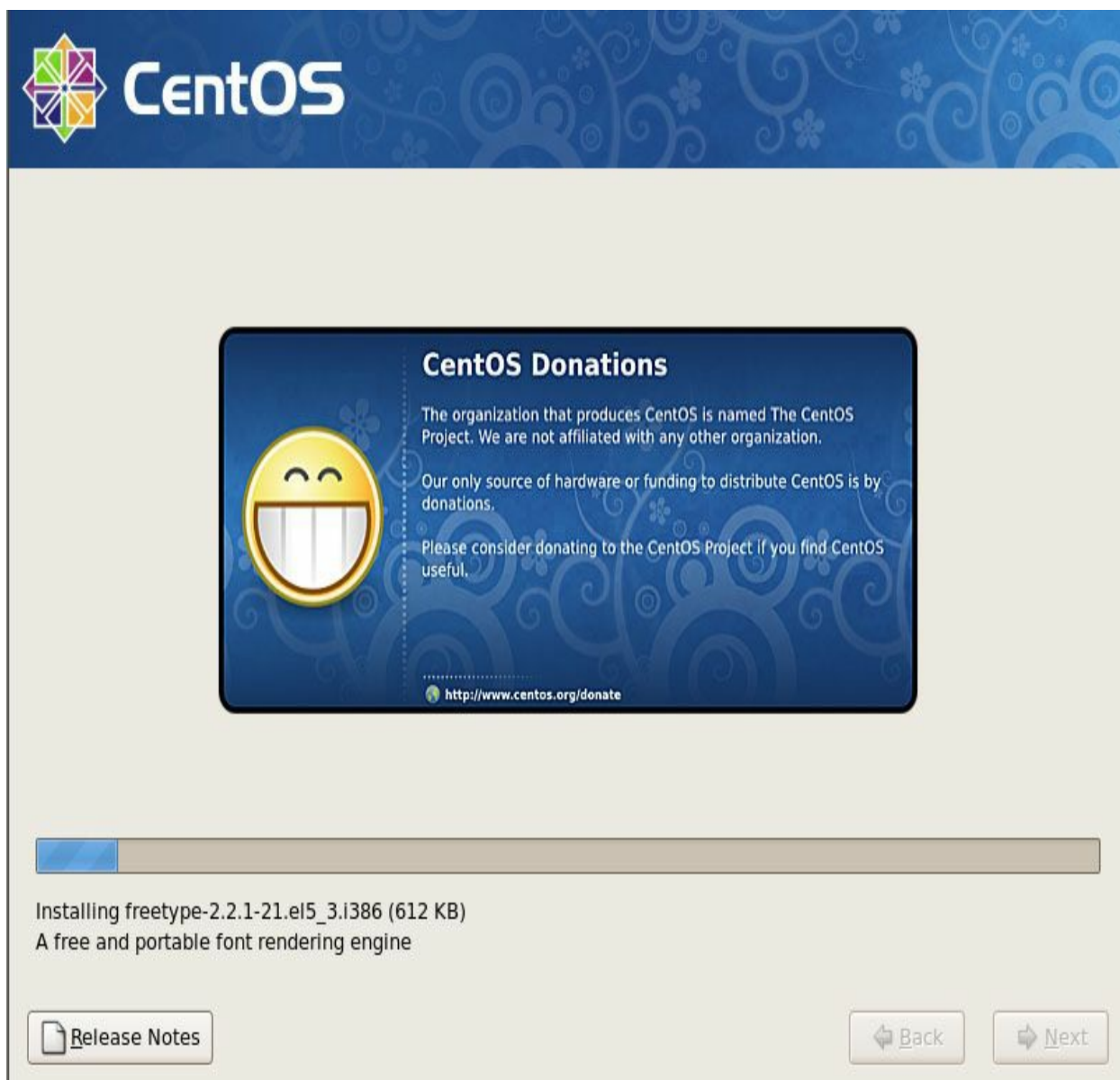


图1-39 正式安装界面

安装结束后，同样需要重启系统，如图1-40所示。

到此，安装过程就已经结束了。

通过以上RedHat和CentOS的安装过程演示，相信大家已经清楚，两种系统的安装过程几乎是一样的，这也再次证明了CentOS和RedHat虽然是两个独立的发行版，但是其实质是一样的。事实上，RedHat在发行的时候都会同时提供二进制代码

和源代码，无论是哪一种方式都可以免费从网络上获得，而CentOS所做的就是将RedHat发行的源代码重新编译，形成一个可用的二进制版本。由于RedHat在某些情况下使用起来不太便利，例如，使用RedHat的官方软件仓库是需要注册RHN的，因此CentOS在重新编译的时候不但保留了RedHat所有的功能，同时还做了不少功能上的优化。

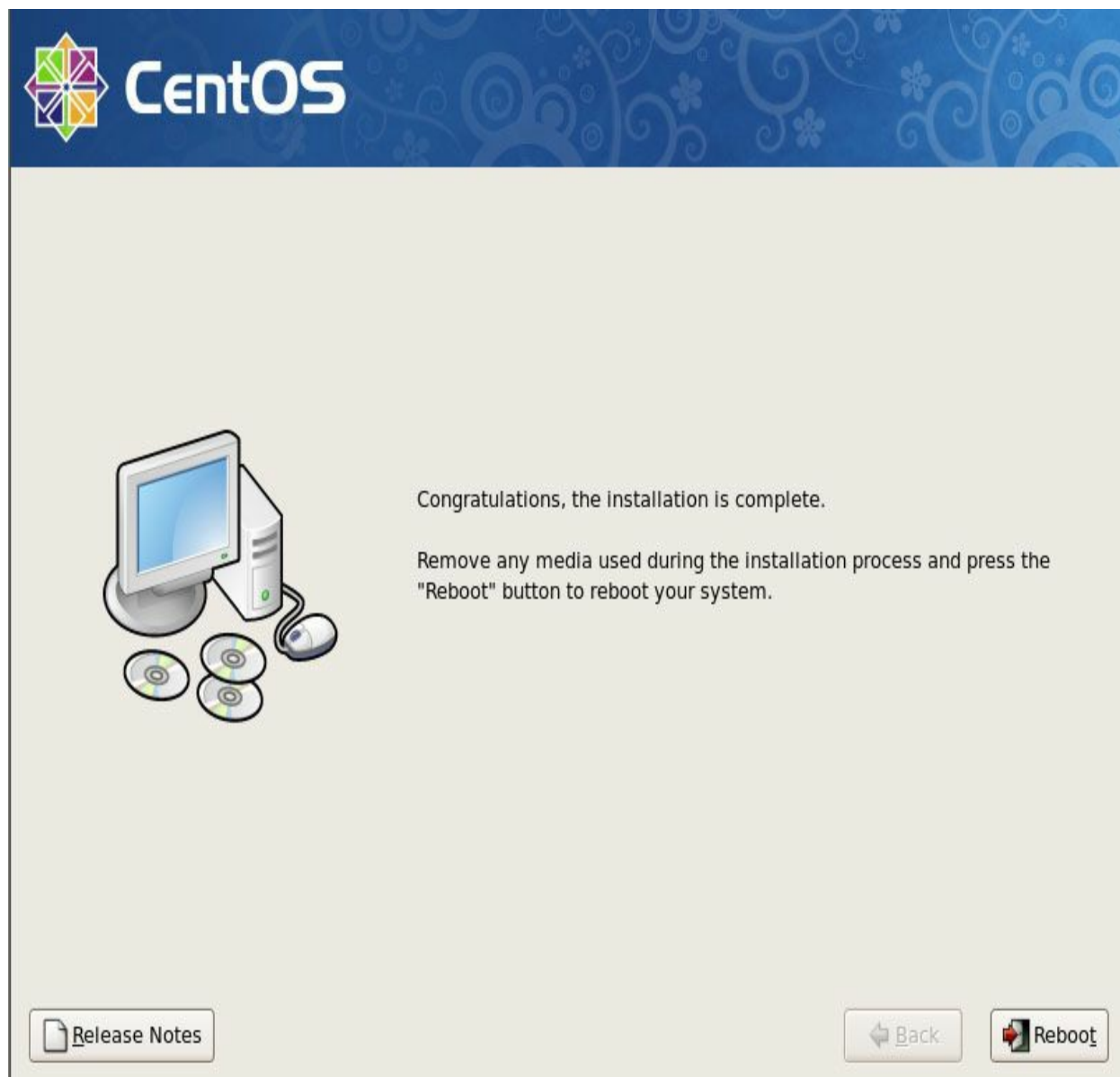


图1-40 安装完成

1.4 系统登录

1.4.1 第一次登录系统的设置

不管是RedHat还是CentOS，在第一次启动时都需要进行“首次启动”的设置，系统称之为First Boot。本节将会继续演示RedHat和CentOS在首次启动时的设置过程。下面就来看看RedHat 5.5的首次启动过程。

第一次启动后，将会进入首次启动的欢迎界面，单击Forward按钮，如图1-41所示。

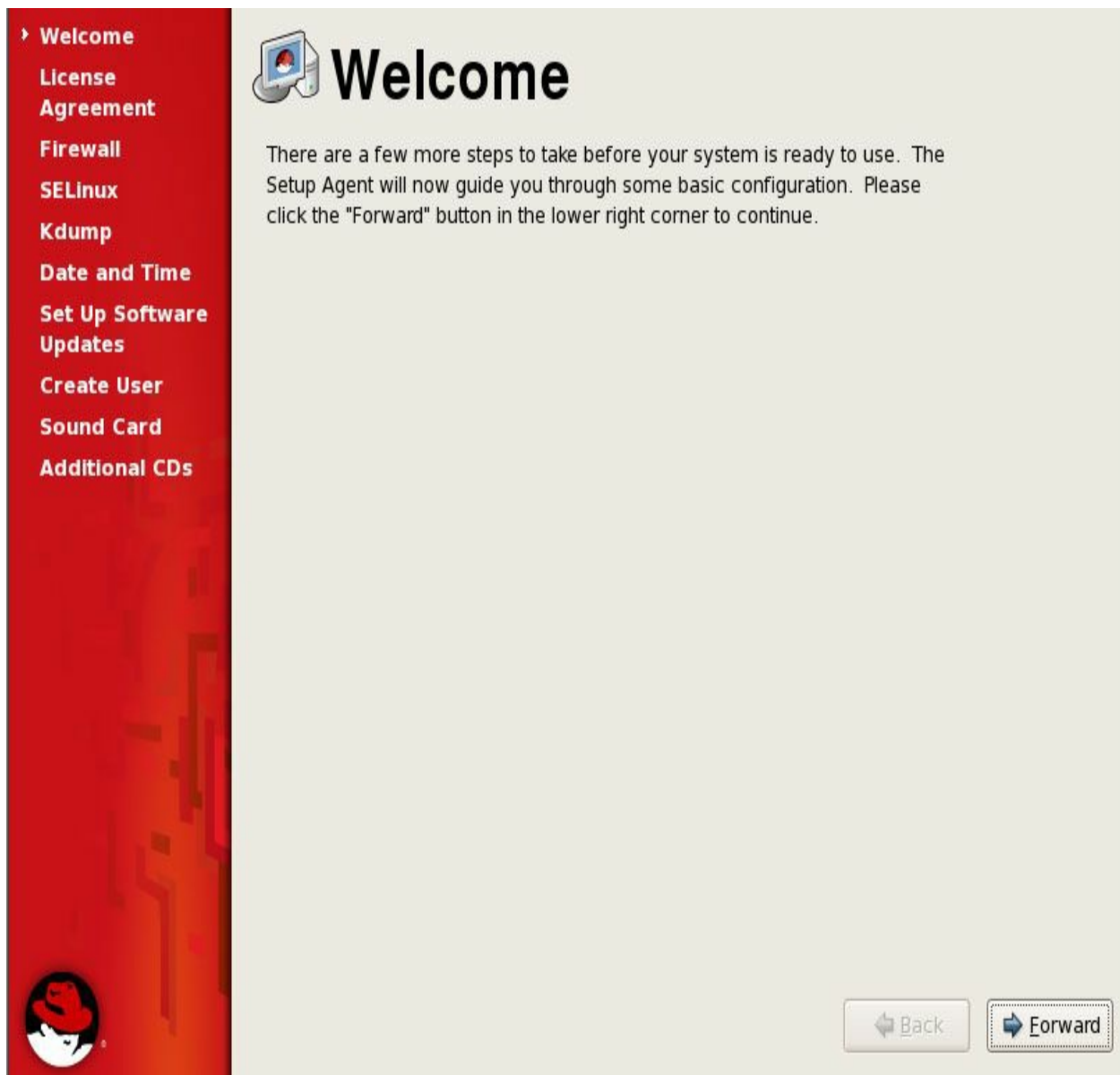


图1-41 首次启动欢迎界面

图1-42所示是RedHat的版权申明，必须选择Yes选项，否则就无法继续了。单击Forward按钮。



图1-42 版权申明

进入防火墙设置。单击Firewall下拉框，选择Disabled关闭防火墙，然后单击Forward按钮。在随后弹出的提示框中，选择Yes选项，如图1-43所示。



图1-43 关闭防火墙

进入SELinux设置。单击SELinux Setting下拉框，选择Disabled，然后单击Forward按钮，在随后弹出的提示框中，选择Yes选项，如图1-44所示。

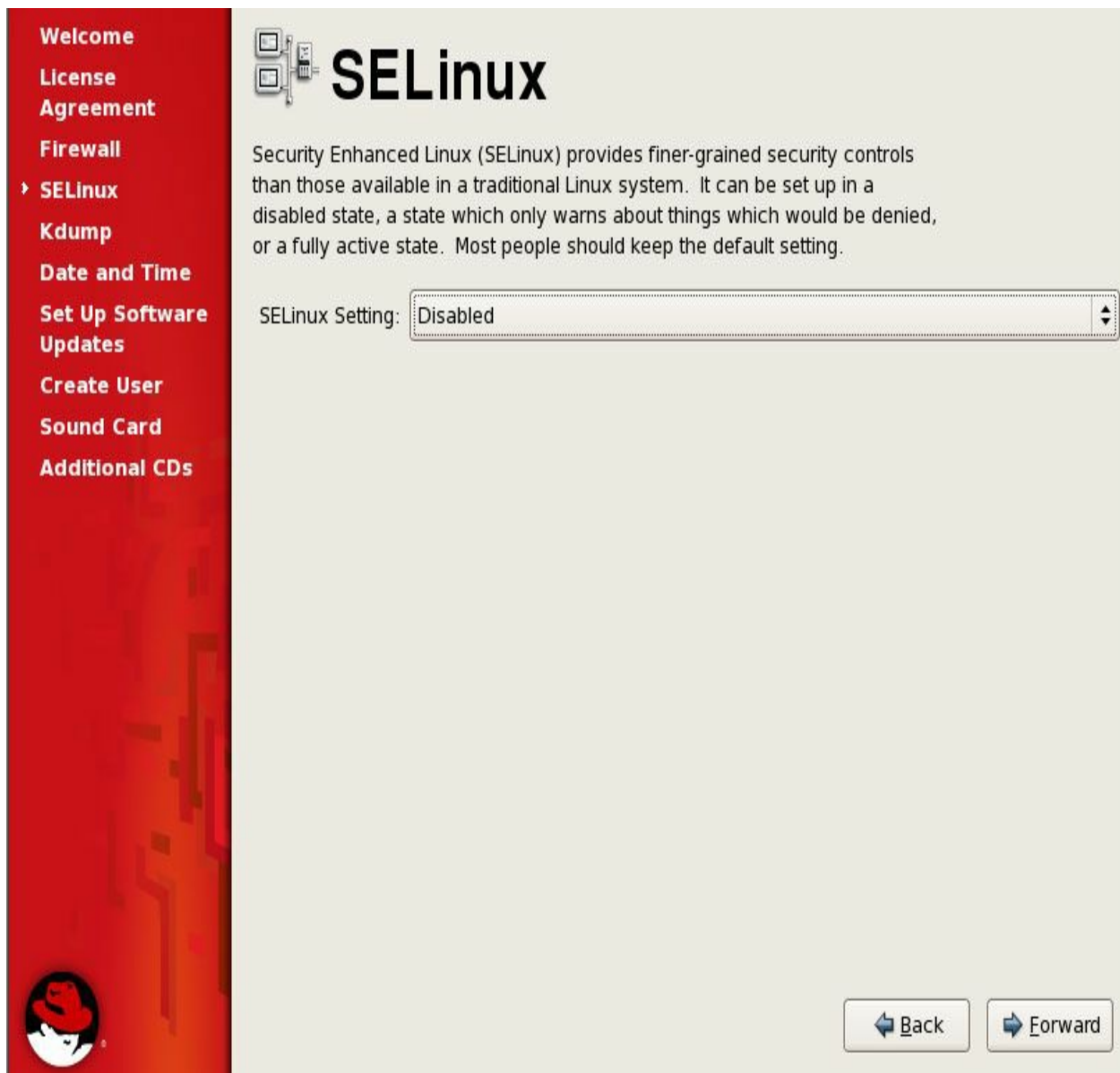


图1-44 关闭SELinux

进入Kdump的设置，默认是关闭的，单击Forward按钮，如图1-45所示。



图1-45 Kdump界面

在如图1-46所示的界面中可设置时间和日期，设置好后，单击Forward按钮。

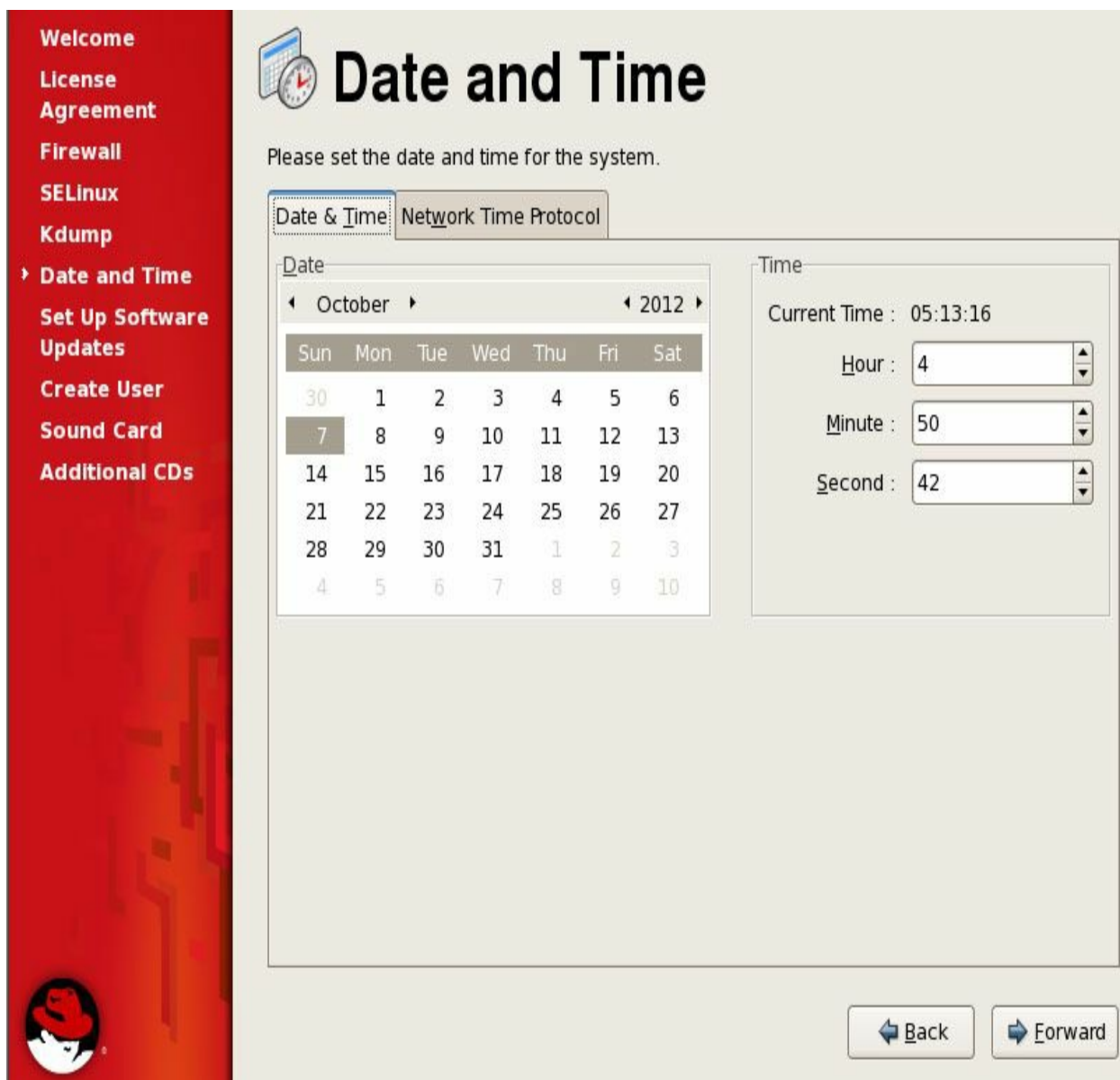


图1-46 时间和日期设置界面

接下来设置RHN（RedHat Network），这里跳过这步，选择No,I prefer register at a later time，然后单击Forward按钮，如图1-47所示。在随后弹出的对话框中，单击No thinks,I will connect later选项。



图1-47 注册RHN

在如图1-48所示的界面中单击Forward按钮，进入下一步。



图1-48 配置完成

系统建议创建一个用户来做一些非管理的任务，不过由于在学习的过程中不少操作需要较高的权限，对于初学者来说，使用非特权用户会在学习过程中遇到意想不到的麻烦。所以这里忽略此步，单击Forward按钮，如图1-49所示。在随后弹出的对话框中选择Continue，确认跳过此步骤。



Welcome

License Agreement

Firewall

SELinux

Kdump


Date and Time

Set Up Software Updates

▶ Create User

Sound Card

Additional CDs



Create User

It is recommended that you create a 'username' for regular (non-administrative) use of your system. To create a system 'username,' please provide the information requested below.

Username:

Full Name:

Password:

Confirm Password:

If you need to use network authentication, such as Kerberos or NIS, please click the Use Network Login button.

Use Network Login...

Back

Forward

图1-49 创建用户界面

设置声卡时，一般直接单击Forward按钮即可，因为谁也不会用服务器来听音乐，如图1-50所示。

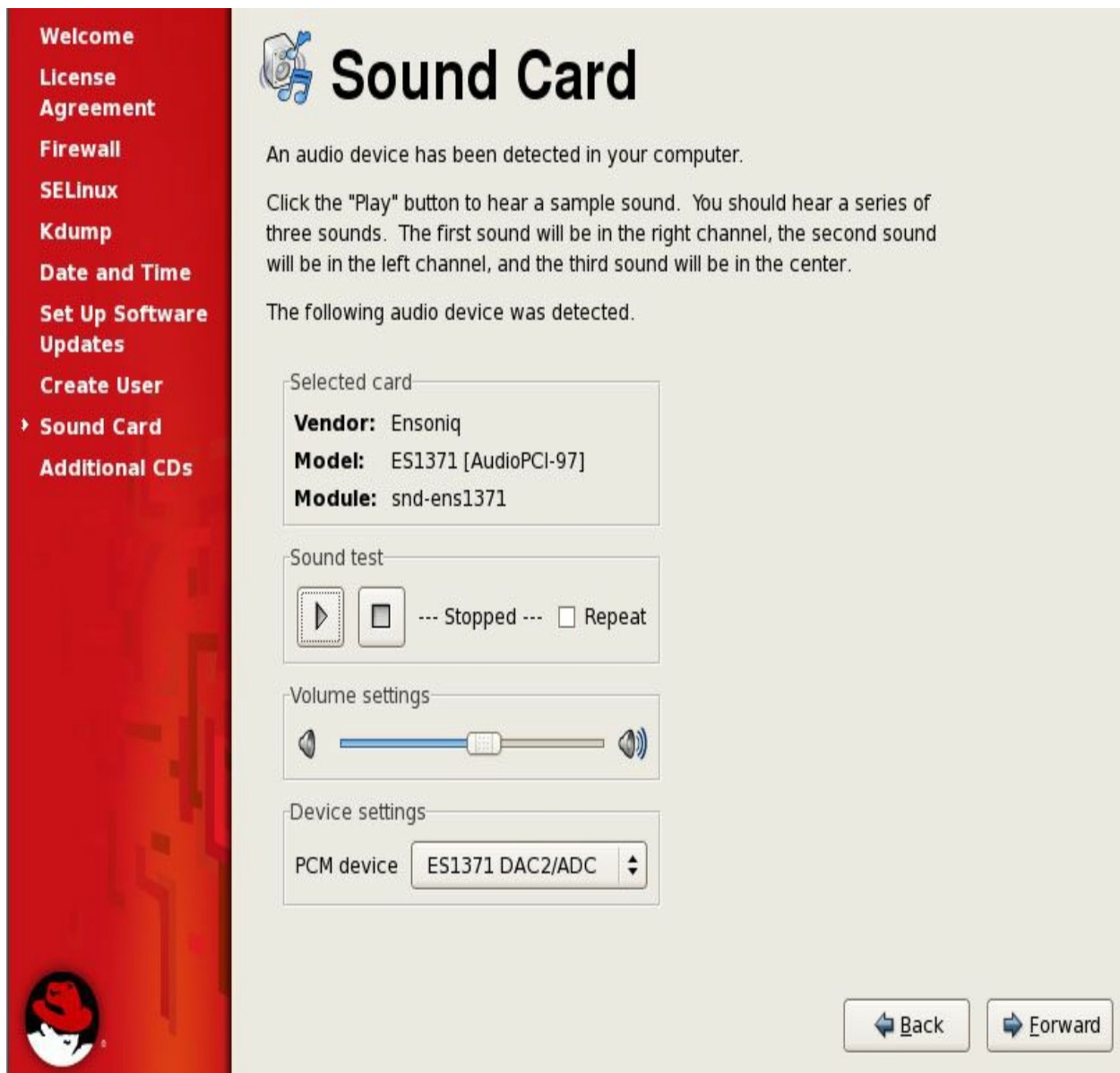


图1-50 声卡检测

如图1-51所示的界面是安装过程中最后一次提供安装软件的机会，只要插入原先的安装光盘就可以选择安装其他包。由于暂时不需要安装特定的软件，这里单击Finish按钮。系统会弹出需要重启对上述配置生效的提示，单击OK按钮后系统将再次重启，至此RedHat的首次启动的设置就结束了。



图1-51 安装过程中最后安装软件的机会

下面再来看看CentOS“首次启动”的设置过程。首先呈现的也是一个欢迎界面，单击Forward按钮，如图1-52所示。



图1-52 欢迎界面

进入防火墙设置界面，单击Firewall下拉框，选择Disabled关闭防火墙，然后单击Forward按钮，在随后弹出的对话框中选择Yes选项，如图1-53所示。



图1-53 设置防火墙

设置SELinux时，单击SELinux Setting下拉框，选择Disabled关闭它，然后单击Forward按钮，如图1-54所示。在随后弹出的对话框中，选择Yes选项。



图1-54 关闭SELinux

接下来要设置日期和时间了，设置好后单击Forward按钮，如图1-55所示。

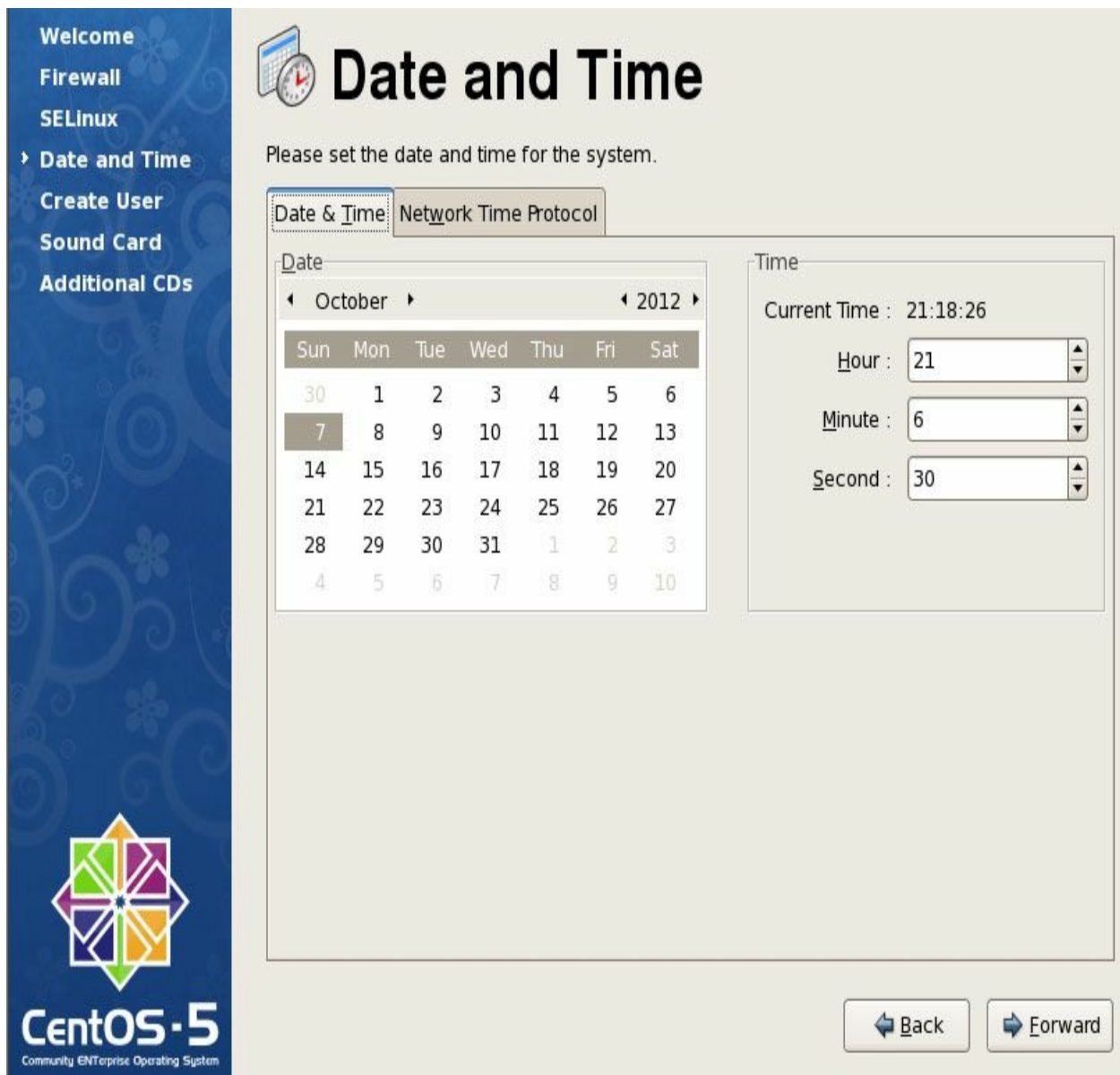


图1-55 日期和时间设置

在如图1-56所示的界面中系统推荐创建一个用户做日常管理，这里忽略直接单击Forward按钮，然后在弹出的对话框中单击Continue按钮。



CentOS-5
Community ENTERprise Operating System

Welcome
Firewall
SELinux
Date and Time
➤ Create User
Sound Card
Additional CDs

Create User

It is recommended that you create a 'username' for regular (non-administrative) use of your system. To create a system 'username,' please provide the information requested below.

Username:

Full Name:

Password:

Confirm Password:

If you need to use network authentication, such as Kerberos or NIS, please click the Use Network Login button.

Use Network Login...

Back Forward

图1-56 创建用户界面

进行声卡设置时，忽略该步骤，单击Forward按钮进入最后一步，如图1-57所示。



图1-57 声卡检测界面

在如图1-58所示的界面中单击Finish按钮以结束全部设置，然后在弹出的对话框中单击OK按钮，系统将会重启以使刚刚设置的所有配置生效。

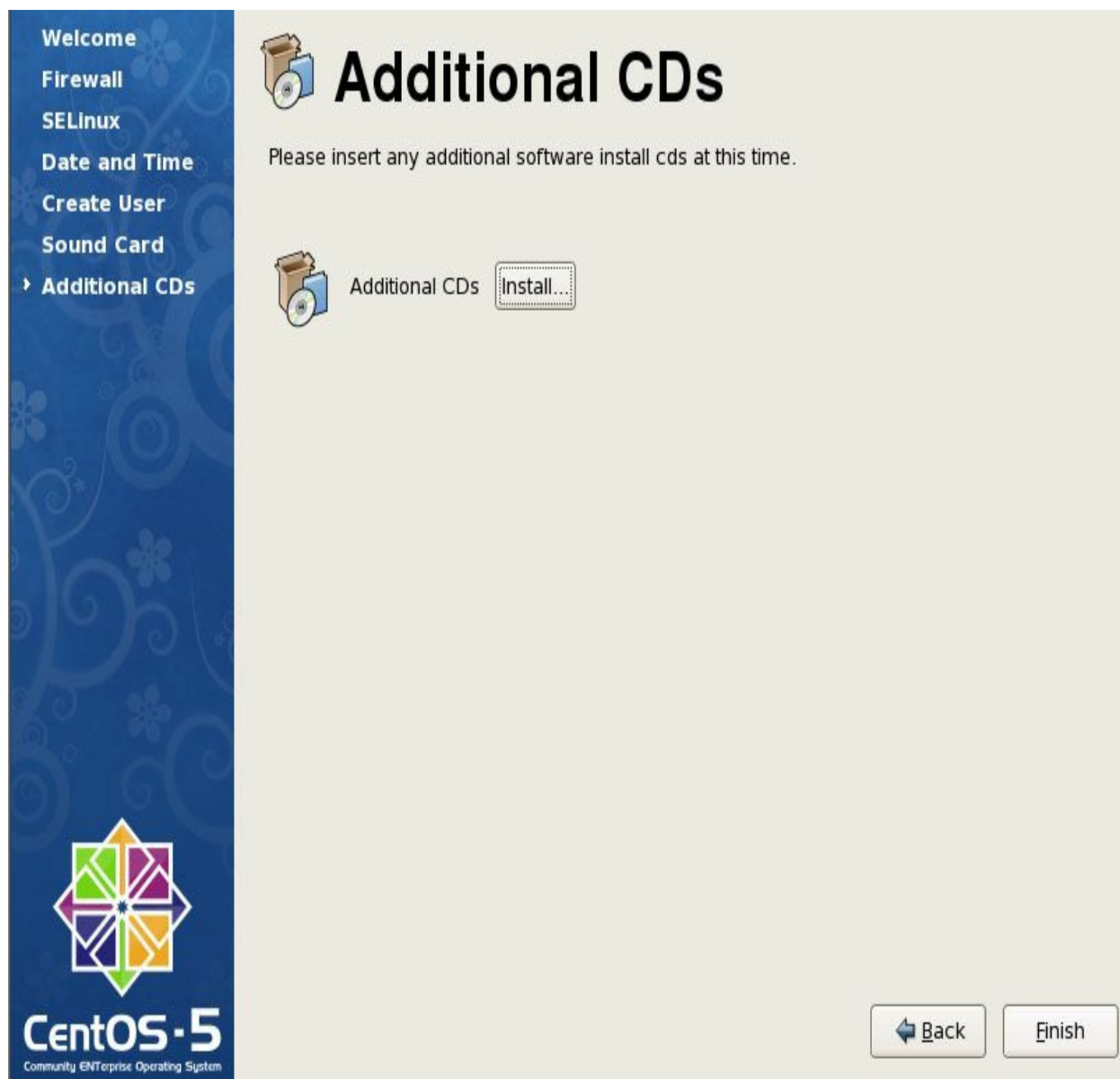


图1-58 结束设置

1.4.2 使用图形模式登录

安装系统并进行了“首次启动”配置后，系统会再次进行重启，最终显示在屏幕前的就是如图1-59所示的登录界面，这个登录界面又称作“登录管理器”。实际上Linux使用了一个X Server的底层程序来提供图形环境，而用户是不能直接与这个X Server交互的，必须通过它运行的图形程序才能进行交互。

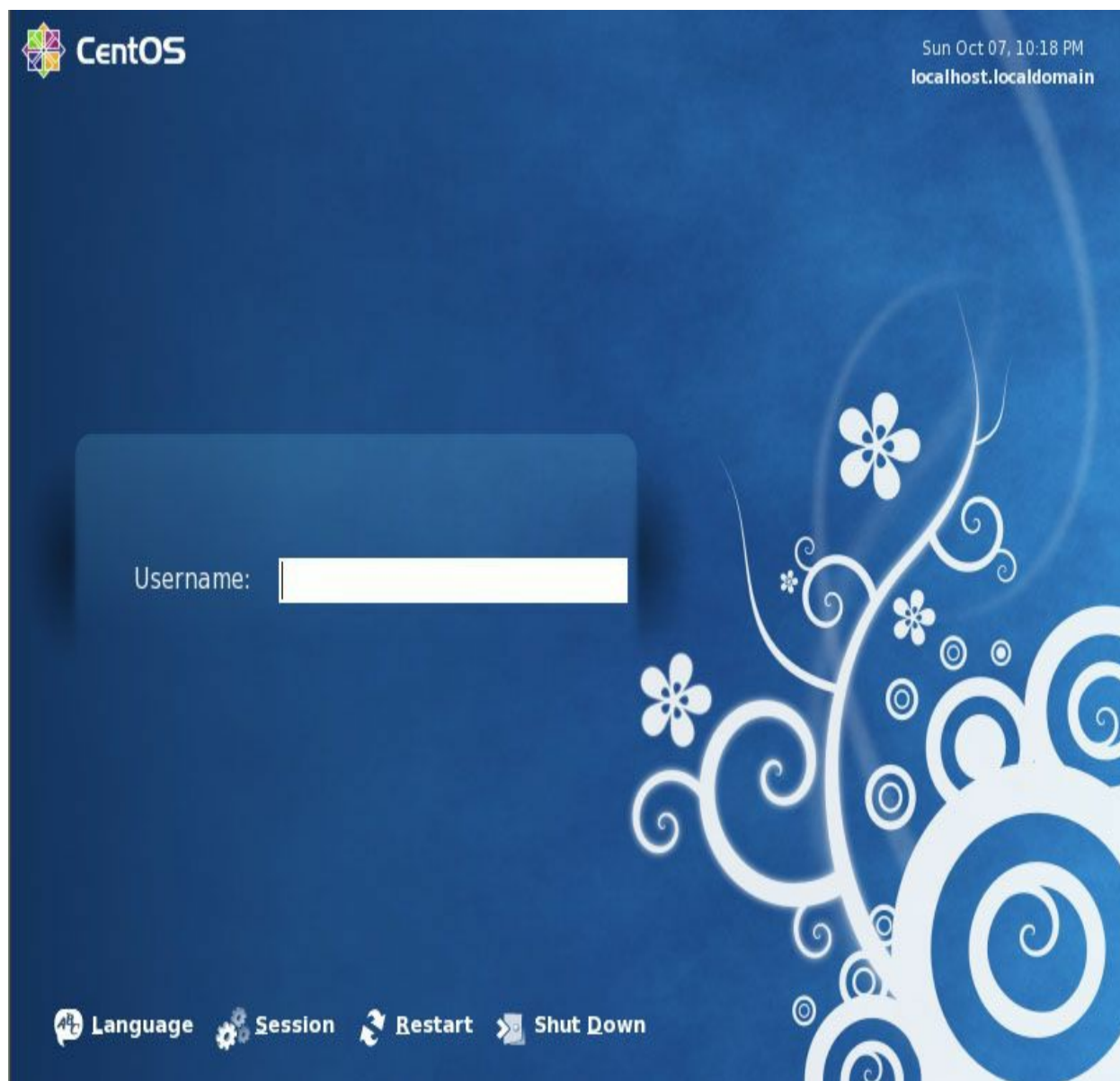


图1-59 登录界面

输入用户名root和正确的密码后，就可以登录进入桌面了。可能有人已经注意到，登录界面的下部有4个选项，分别是Language、Session、Restart、Shut Down。

单击Language，可以看到有各种语言，有可能有一些呈现方块状的乱码文字，那是因为缺少相关文字的文字包，导致字体显示不正常，但是应该不影响大家了解Language的作用就是选择不同的语言作为登录后的默认语言。

单击Session，可以看到系统提供了3种登录方式，即Gnome、KDE、Failsafe，这些都是常用的图形化登录方式。其实这些都是Linux下的桌面环境，大家可以根据个人喜好选择。

登录后桌面上默认会有3个图标，如图1-60所示，分别是Computer、root's Home和Trash，分别类似于Windows下的“我的电脑”、“我的文档”、“回收站”。左上角有3个面板，分别是Applications、Places、System，其中Applications中放置的是应用程序，类似Windows下的“所有程序”；Places主要是各种存储设备；而System是系统配置相关的部分，大家可以单击一下看看都有什么。桌面的右下角有4个方框，这是图形界面下的虚拟桌面，可以在不同的虚拟桌面上运行不同的应用，相信这个不难理解。



图1-60 Gnome桌面

在图形界面下，最有用的当属gnome-terminal了，打开它的方式有两种。第一种，如图1-61所示，依次在图形界面上点选 **Applications** → **Accessories** → **Terminal**，打开图形终端；第二种，在桌面上右击，然后点选 **Open Terminal**，如图1-62所示。

退出图形登录的方法也很简单，在 **System** 中选择 **Log Out root** 即可。

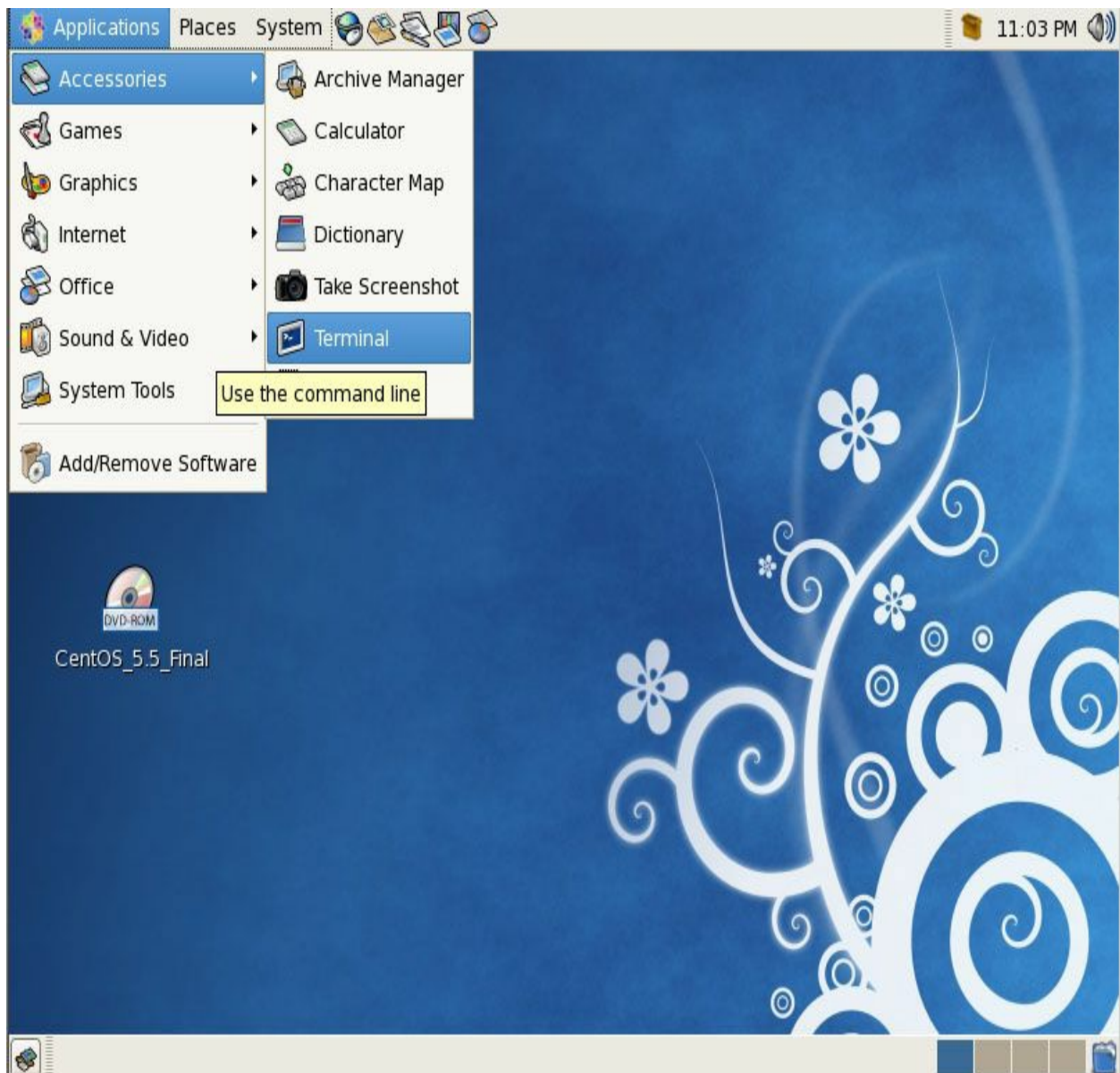


图1-61 终端启动方式一

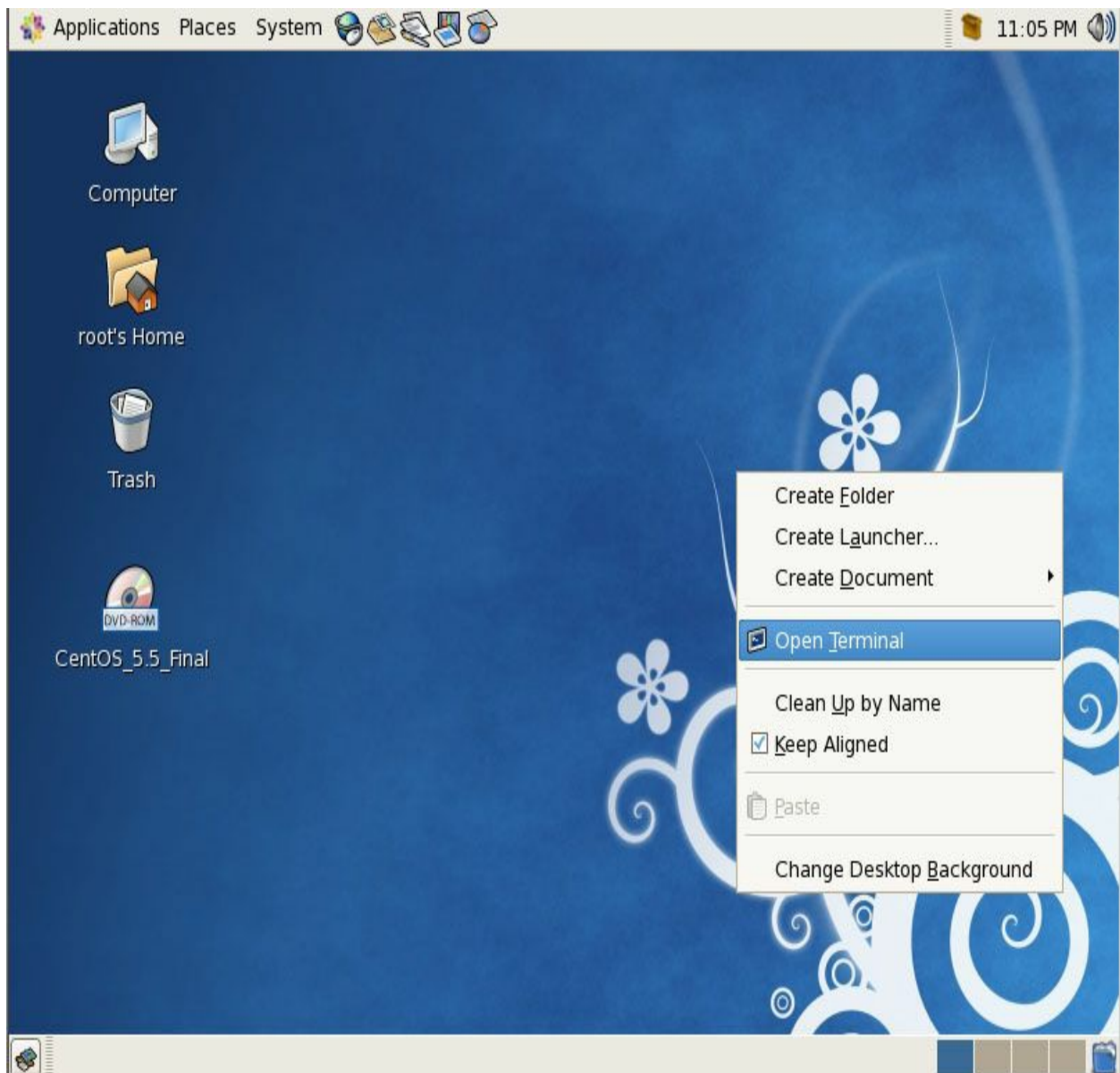


图1-62 终端启动方式二

RedHat和CentOS都默认使用Gnome作为桌面环境，不过说到底，这些桌面环境都只是Linux环境下的软件，所以对桌面的使用方法不是学习Linux的重点，所以笔者也不准备对图形界面做更多的叙述。

1.4.3 使用终端模式登录

终端模式又称为命令行模式或字符模式，默认情况下Linux提供6个终端，可以使用组合键Ctrl+Alt+F1进入第一个终端，使用组合键Ctrl+Alt+F2进入第二个终端，其他终端的组合键以此类推。实际上，终端又叫tty，Linux系统定义了6个tty，分别从tty1到tty6。tty是Teletype的简写，Teletype是最早出现的一种终端设备，很像电传打字机。在Linux系统中，在特殊文件目录/dev下有一些文件与之对应，比如/dev/tty1、/dev/tty2等，从tty1到tty6又称为虚拟终端。如果想回到桌面模式，只需要使用组合键Ctrl+Alt+F7即可。

如果系统设置默认启动的时候不启动图形界面（下一小节中我们会提到系统中的一个重要的概念：runlevel，当runlevel为3时，则不启动图形界面），在这个情况下，tty7是不可用的，这时候要想从终端字符界面进入图形界面就需要使用startx这个命令了。命令如下所示（当然是否能启用图形桌面还取决于系统是否正确地安装了图形桌面系统）。

```
[root@localhost ~]# startx
```

如果现在在字符登录界面，默认屏幕上会显示如下内容：

```
CentOS release 5.5 (Final)
Kernel 2.6.18-194.el5 on an i686
Localhost login:root
Password:
Last login: Tue Oct  9 22:07:00 2012
[root@localhost ~]#
```

其中，第一行是发行版的名称（CentOS）和版本号

(5.5)；第二行是内核版本(2.6.18-194.el5)，以及当前运行的硬件平台(i686)；第三行是主机名(localhost)，login后面等待用户输入，这里输入“root”；第四行等待输入root用户的密码；第五行是当成功登录时，系统会显示出该用户上次成功登录的时间；第六行显示登录成功后用户和主机名以及所在的目录，“~”是用户home目录(又叫“用户家目录”)的简写。最后的“#”是一个提示符，出现“#”说明目前的用户是有超级权限的root用户，而一般用户的提示符是“\$”。现在已经登录到字符界面中了。

读者或许已经注意到，登录前字符终端上打印出来了一些系统信息(第一行和第二行)，它们实际上来自系统中的一个配置文件。为了让大家理解Linux系统中“一切皆文件”的概念，同时提高大家对Linux系统的兴趣，让我们一起来做个小实验。首先使用如下命令编辑文件：

```
[root@localhost ~]# vi /etc/issue
```

在随后出现的界面中，按住键盘上的Shift+G组合键(也就是输入大写字母G)，再按字母o键，接着输入“Hello, Welcome to Linux”，之后按Esc键，然后按一下冒号键，在冒号后面输入字母x，按回车键，最后在窗口中输入命令exit。看看现在的登录界面与之前有什么不一样？做了这个实验后能得到什么结论呢？还有，刚刚大家其实已经用了一部分Linux下强大的字符编辑器vi了，关于此编辑器更详细的使用方法后面会专门讲解。

值得提醒的是，在平时的工作中，当你登录到系统中进行操作后，一定要记得在离开终端前要输入exit命令退出当前的登录用户，防止他人利用该账户进行操作而造成麻烦。

1.4.4 开始学习使用Linux的命令

相信读者或多或少都知道，对Linux的管理大多使用的是命令行模式，这是为什么呢？命令行界面有很多优点，尤其是它的高效灵活让Linux的管理非常有效率。但是命令行使用起来并不简单，必须长期使用才能熟能生巧。本节将通过几个常见的命令来介绍一下命令的一般使用方法。

1.显示日期：date

```
[root@localhost ~]# date
Thu Oct 11 23:05:54 CST 2012
```

上面显示的时间是：星期四，10月11日，23点5分54秒，CST时区，2012年。这里要说明的是，Linux下的命令是严格区分大小写的。例如，把date写成DATE，就会提示command not found，也就是没有这个命令，如下所示：

```
[root@localhost ~]# DATE
-bash: DATE: command not found
```

当然，date命令后也可以加上一些“参数”来调整命令显示内容，如下所示：

```
[root@localhost ~]# date +%Y%m%d
20121011
```

上面显示的是2012年10月11日。date命令本身还有其他的一些参数，通过不同的参数可以显示出不同的内容。命令和参

数之间使用一个或者多个空格隔开。

2.列出目录内容：ls

```
[root@localhost ~]# ls
anaconda-ks.cfg  Desktop  install.log  install.log.syslog
```

使用root登录系统后，使用ls命令可以列出当前目录下的内容，上面的命令显示了anaconda-ks.cfg、Desktop、install.log、install.log.syslog四个内容。不过看起来好像没什么区别，让我们在这个命令后加一个参数试试。

```
[root@localhost ~]# ls -l
total 60
-rw----- 1 root root  954 Oct  7 21:02 anaconda-ks.cfg
drwxr-xr-x 2 root root 4096 Oct  7 22:53 Desktop
-rw-r--r-- 1 root root 30975 Oct  7 21:02 install.log
-rw-r--r-- 1 root root  4492 Oct  7 20:59 install.log.syslog
```

从所显示内容的第一列可以看到，其实Desktop不同于其他3个，注意到Desktop所在行的第一个字母是d，这说明它是一个目录（在后面会详细讲到该位上不同的字符所代表的不同含义），而其他3个都是普通文件。通过这个例子可以知道，ls-l的作用是详细显示当前目录下的所有文件。

如果只是想详细显示其中一个文件，那么该怎么做呢？只要加上需要显示的文件就可以了。这说明ls命令除了-l选项之外，还可以在后面再加参数。比如下面是添加了anaconda-ks.cfg参数：

```
[root@localhost ~]# ls -l anaconda-ks.cfg
-rw----- 1 root root 954 Oct  7 21:02 anaconda-ks.cfg
```

3.显示文件内容： cat

anaconda-ks.cfg是一个文本文件，那么里面的内容是什么呢？可以使用cat命令来显示。

```
[root@localhost ~]# cat anaconda-ks.cfg
# Kickstart file automatically generated by anaconda.
.....(
略去内容).....
```

上面给大家展示了几个命令的基本使用方式。一般来说，命令在使用中有以下几种方式：

- 部分命令后面可以直接回车。
- 部分命令后面可以跟上特定的“选项”作为该命令的参数。
- 不同的命令所能跟的参数以及参数的个数一般不同。

1.5 系统启动流程

1.5.1 系统引导概述

为了更好地了解Linux系统的运行原理，非常有必要了解系统启动的流程。实际上，这也是学习Linux应知应会的内容，在很多Linux系统工程师的职位面试中都会被问及。

来想象一下台式机的启动过程，相信大家都有这样的经验和体会。在按开机电源后，会听到机箱内发出“滴”的一声，接着屏幕上开始打印出一些字符，然后开始显示出图形界面，最后屏幕上会显示需要输入用户名、密码的登录界面。其实，不管是Linux还是Windows，从用户感官上的体验而言，顺序都是大同小异的。本节将详细描述Linux环境下的启动流程，起点是从按下计算机的电源键开始。

首先，计算机会加载BIOS，这是计算机上最接近硬件的软件，各家主板制造商都会开发适合自己主板的BIOS，而BIOS中一项很重要的功能就是对自身的硬件做一次健康检查，只有硬件没有问题，才能运行软件，记住，操作系统也是一种软件。这种通电后开始的自检过程被称为“加电自检”，英文中称为Power On Self Test，简称POST。如果所有的硬件自检通过，一般都会发出一次“滴”的短声提示，说明硬件一切正常。

机器自检通过后，下面就要引导系统了。这个动作是BIOS设定的，BIOS默认会从硬盘上的第0柱面、第0磁道、第一个扇区中读取被称为MBR的东西，即主引导记录。一个扇区的大小是512字节，存放的内容是一段引导程序和分区信息，其中引导程序部分占用446字节，另外64字节是磁盘分区表DPT，最后两字节是MBR的结束位。这512字节的空间内容是由专门的分区程序产生的，比如说Windows下的fdisk.exe，或

者Linux下的fdisk命令，所以它不依赖于任何操作系统，而MBR中的引导程序也是可以修改的，所以可以利用这个特性实现多操作系统共存。由于RedHat、CentOS默认会使用Grub作为其引导操作系统的程序，而Grub本身又比较大，所以常见的方式是在MBR中写入Grub的地址，这样系统实际会载入Grub作为操作系统的引导程序。

经过了上面的步骤，第三步就是顺理成章地运行Grub了。Grub最重要的功能就是根据其配置文件加载kernel镜像，并运行内核加载后的第一个程序/sbin/init，这个程序会根据/etc/inittab来进行初始化的工作。其实这里最重要的就是根据文件中设定的值来确定系统将会运行的runlevel，默认的runlevel定义在“id:3:initdefault:”中，其中的数字3说明目前的运行级别定义为3（这里提到了runlevel的概念，将在后面详细讲解）。

第四步，Linux将根据/etc/inittab中定义的系统初始化配置si::sysinit:/etc/rc.d/rc.sysinit执行/etc/rc.sysinit脚本，该脚本将会设置系统变量、网络配置，并启动swap、设定/proc、加载用户自定义模块、加载内核设置等。

第五步是根据第三步读到的runlevel值来启动对应的服务，如果值为3，就会运行/etc/rc3.d/下的所有脚本，如果值为5，就会运行/etc/rc5.d/下的所有脚本。

第六步将运行/etc/rc.local，第七步会生成终端或X Window来等待用户登录。

1.5.2 系统运行级别

前一节多次提到了runlevel这个词，但是runlevel究竟是什么呢？我们说Linux默认有7个运行级，从运行级0到运行级6，每一个运行级所对应的含义如下：

运行级0：关机。

运行级1：单用户模式，系统出现问题时可使用这种模式进入系统维护，典型的使用场景是在忘记root密码时可进入此模式修改root密码。

运行级2：多用户模式，但是没有网络连接。

运行级3：完全多用户模式，这也是Linux服务器最常见的运行级。

运行级4：保留未使用。

运行级5：窗口模式，支持多用户，支持网络。

运行级6：重启。

任何时候Linux只能在一种runlevel下运行。那么不同的runlevel之间到底有什么区别呢？上一节中提到，系统在启动的过程中会根据/etc/inittab中的设定读取runlevel的数值X，并相应地读取和运行/etc/rcX.d（X代表0~6）下所有的脚本。看一下/etc/rc3.d中的内容：

```
[root@localhost ~]# ll /etc/rc3.d/
total 288
.....(
略去内容).....
lrwxrwxrwx 1 root root 15 Oct 7 20:52 K15httpd -
```

```
> ../init.d/httpd
lrwxrwxrwx 1 root root 13 Oct 7 20:55 K20nfs -
> ../init.d/nfs
.....(
略去内容).....
lrwxrwxrwx 1 root root 18 Oct 7 20:50 S08iptables -
> ../init.d/iptables
lrwxrwxrwx 1 root root 17 Oct 7 20:52 S10network -
> ../init.d/network
.....(
略去内容).....
```

注意看每行中第9列的内容，分别是以K或S开头、后跟两位数字、再接服务名的文件，其实它们链接的是上层init.d目录中的服务脚本。系统在启动过程中，会首先运行以K开头的脚本，全部运行完毕后再运行以S开头的脚本，在运行所有K开头的脚本时，又会严格按照K后面的数字大小依次来运行，也就是数字小的先运行，数字大的后运行。同样，在运行S开头的脚本时，也是按照这个原则进行的，即先运行数字小的脚本，再运行数字大的脚本。K和S的意思分别是停止（kill）和启动（start），只要定义好不同运行级需要启动和停止的服务，就可以让系统在不同的运行级下启动和关闭不一样的服务。再来对比一下/etc/rc1.d下的关于network项内容：

```
[root@localhost ~]# ll /etc/rc1.d/
total 288
.....(
略去内容).....
lrwxrwxrwx 1 root root 17 Oct 7 20:52 K90network -
> ../init.d/network
.....(
略去内容).....
```

在运行级为1的时候，network是在开机启动的过程中被关闭的（K90network），而在运行级为3的时候，network则是被开启的（S10network）。

1.5.3 服务启动脚本

上节在介绍Linux运行级时，谈到在Linux启动过程中会使用K或S开头的脚本关闭或启动相关服务，那么这是怎么做到的呢？本节将通过一个脚本帮助大家理解。当然因为这里还没有讲到Shell编程的内容，所以只做非常简单的讲解。

```
#!/bin/bash
#
# 一个bash
# 脚本开始的标记，必须是用“#!/bin/bash
# 开头，含义是提示系统在运行该脚本时使用
# /bin/bash
# 作为执行该文件的解释器
# /etc/rc.d/init.d/atd
#
# 说明自己的绝对路径
# Starts the at daemon
#
# chkconfig: 345 95 5
#345
# 是说在运行级是345
# 的时候，默认开启atd
# ，也就是Start
#95
# 是说明当默认设置为on
# 的时候，运行优先级定为95
#5
# 是说明当默认设置为off
# 的时候，停止优先级定为5
# description: Runs commands scheduled by the at command at t
# specified when at was run, and runs batch commands when
# average is low enough.
# processname: atd
# Source function library.
# . /etc/init.d/functions
#
# 使用“.
# 命令包含文件，可以使用/etc/init.d/functions
# 中定义的函数
# pull in sysconfig settings
[ -f /etc/sysconfig/atd ] && . /etc/sysconfig/atd
```



```

test -x /usr/sbin/atd || exit 0
RETVAL=0
#
#       See how we were called.
#
prog="atd"
start() {
    # Check if atd is already running
    if [ ! -f /var/lock/subsys/atd ]; then
        echo -n "Starting $prog: "
        daemon /usr/sbin/atd $OPTS && success || failure
        RETVAL=$?
        [ $RETVAL -eq 0 ] && touch /var/lock/subsys/atd
        echo
    fi
    return $RETVAL
}
#
定义start
函数
stop() {
    echo -n "Stopping $prog: "
    killproc /usr/sbin/atd
    RETVAL=$?
    [ $RETVAL -eq 0 ] && rm -f /var/lock/subsys/atd
    echo
    return $RETVAL
}
#
定义stop
函数
restart() {
    stop
    start
}
#
定义restart
函数，实际调用时，先执行stop
函数后执行start
函数
reload() {
    restart
}
#
定义reload
函数，实际调用时，就是执行restart
函数
status_at() {

```

```

        status /usr/sbin/atd
    }
    #
    定义status_at
    函数，实际调用时，是调用/etc/init.d/functions
    中定义的函数status
    ,
    参数为/usr/sbin/atd
    , 也就是查询atd
    的运行状态
    case "$1" in
    start)
        start
        ;;
    stop)
        stop
        ;;
    reload|restart)
        restart
        ;;
    condrestart)
        if [ -f /var/lock/subsys/atd ]; then
            restart
        fi
        ;;
    status)
        status_at
        ;;
    *)
        echo $"Usage: $0 {start|stop|restart|condrestart|stat
        exit 1
    esac
    exit $?
    exit $RETVAL

```

上面的脚本实际上是/etc/init.d/atd中的内容，我在脚本中做了一些注释来简单讲解脚本的处理过程。当atd设置为启动时，将会在对应的/etc/rcX.d（X代表0~6）目录下显示：S95atd->../init.d/atd，系统根据第一个字母S判定atd需要启动，然后会调用命令/etc/init.d/atd start；当atd设置为关闭时，将会在对应的/etc/rcX.d目录下显示：K05atd->../init.d/atd，系统根据第一个字母K判定atd需要关闭，然后调用命令/etc/init.d/atd stop，这样就实现了对atd的启停控制，其他服务也是同样的原

理。

1.5.4 Grub介绍

在之前的系统引导概述中，相信大家已经看到Grub这个词了，它的全称为Grand Unified Bootloader，也是GNU赞助的项目之一，事实上Grub可以引导多个操作系统。早先Linux的引导程序是lilo，含义为Linux Loader，这是ext2文件系统中特有的引导程序，现在基本上已经不再使用了。

在之前的系统启动流程中提到，计算机在启动时，BIOS默认会从硬盘上的第0柱面、第0磁道、第一个扇区中读取512字节的数据来引导系统启动，但是Grub这个程序远远大于512字节，这一个扇区又如何能够载下Grub所有的内容呢？为了解决这个问题，实际上Grub的启动是分成两段完成的。第一段以stage1作为主引导程序，它的主要任务是定位和装载第二段引导程序，并转交控制权，即stage2。Grub目录中的内容如下：

```
[root@localhost grub]# cd /boot/grub/
[root@localhost grub]# ls -l
total 257
-rw-r--r-- 1 root root    63 Oct  7 21:02 device.map
-rw-r--r-- 1 root root  7584 Oct  7 21:02 e2fs_stage1_5
-rw-r--r-- 1 root root  7456 Oct  7 21:02 fat_stage1_5
-rw-r--r-- 1 root root  6720 Oct  7 21:02 ffs_stage1_5
-rw----- 1 root root   573 Oct  7 21:02 grub.conf
-rw-r--r-- 1 root root  6720 Oct  7 21:02 iso9660_stage1_5
-rw-r--r-- 1 root root  8192 Oct  7 21:02 jfs_stage1_5
lrwxrwxrwx 1 root root    11 Oct  7 21:02 menu.lst -
> ./grub.conf
-rw-r--r-- 1 root root   6880 Oct  7 21:02 minix_stage1_5
-rw-r--r-- 1 root root   9248 Oct  7 21:02 reiserfs_stage1_5
-rw-r--r-- 1 root root 55808 Mar 13 2009 splash.xpm.gz
-rw-r--r-- 1 root root    512 Oct  7 21:02 stage1
-rw-r--r-- 1 root root 104988 Oct  7 21:02 stage2
-rw-r--r-- 1 root root   7072 Oct  7 21:02 ufs2_stage1_5
-rw-r--r-- 1 root root   6272 Oct  7 21:02 vstafs_stage1_5
-rw-r--r-- 1 root root   8904 Oct  7 21:02 xfs_stage1_5
```

注意一下，有一个stage1的文件，大小为512字节，正好是一个扇区的大小。其实这不是一个巧合，stage1确实是MBR的一个副本。还可以看到有很多文件是以stage1_5结尾的，事实上这些文件是各种文件系统的驱动文件，当stage1从不同的文件系统中读取stage2时将用到这些驱动文件。

对Grub的配置可以通过修改Grub的配置文件完成，一般配置文件为/boot/grub/grub.conf。修改后的配置将直接影响下次引导时的行为。下面是系统安装过程中自动生成的配置：

```
# grub.conf generated by anaconda
#
# Note that you do not have to rerun grub after making change
# NOTICE:  You have a /boot partition.  This means that
#           all kernel and initrd paths are relative to /boot/
#           root (hd0,0)
#           kernel /vmlinuz-version ro root=/dev/sda3
#           initrd /initrd-version.img
#boot=/dev/sda
default=0
timeout=5
splashimage=(hd0,0)/grub/splash.xpm.gz
hiddenmenu
title CentOS (2.6.18-194.el5)
        root (hd0,0)
                                kernel      /vmlinuz-2.6.18-
194.el5 ro root=LABEL=/ rhgb quiet
                                initrd /initrd-2.6.18-194.el5.img
```

其中，default=0的含义是默认从第一个title处启动。这里的配置文件中只有一个title项，但是如果还有第二个title项，则可以配置默认从第二个title处引导系统，只要把default改为1就可以了（注意这里的计数是从0开始的）。

timeout=5的含义是显示这个title项时，同时有5秒倒计时，5秒内可以按回车键提前从默认的启动项中启动，也可以按上下键立即停止倒计时，选定一个title，然后按回车键确认从选

定的title中启动。也可以选定某一个title后，按e键进入编辑模式，这样可以即时对Grub进行配置，但是这时的配置并不会写入配置文件中，而只是当时生效。

splashimage是指定启动时的背景图像。如果系统使用的是sata磁盘，则命名规则为：第一块磁盘是sda，第二块磁盘是sdb，以此类推。对磁盘进行分区后的分区命名规则是，第一个磁盘的第一个分区是sda1，第一个磁盘的第二个分区是sda2，第二个磁盘的第一个分区是sdb1，第二个磁盘的第二个分区是sdb2。而Grub使用hd0代表第一块磁盘，而这里（hd0,0）的含义是第一块磁盘的第一个分区。所以（hd0,0)/grub/splash.xpm.gz的绝对路径就是/boot/grub/splash.xpm.gz，这是一个压缩文件，Grub在启动时会自动对该文件做解压缩。

hiddenmenu是设置启动时是否显示菜单。

title是系统引导时显示的名字，这只是一种识别性的文字，可以任意修改。文件的最后3行是相互关联的，第一行root（hd0,0）参数指定了内核放置的分区；第二行kernel/vmlinuz-2.6.18-194.el5 ro root=LABEL=/rhgb quiet指定了内核的路径，表示内核是（hd0,0）分区中的vmlinuz-2.6.18-194.el5文件，ro root=LABEL=/rhgb quiet是启动内核时向内核传入的参数；最后一行initrd/initrd-2.6.18-194.el5.img指定了initrd文件的路径是（hd0,0）中的initrd-2.6.18-194.el5.img文件。

这里第一次提到initrd文件，其英文含义是boot loader initialized RAM disk，也就是boot loader用于初始化的内存磁盘，是系统启动时的临时文件系统，kernel通过读取initrd来获得各种可执行文件和设备驱动，并挂载真实的文件系统，然后卸载这个临时文件系统。在桌面或者Linux服务器中，initrd文件只是一个临时的文件系统，其生命周期很短，只会用作挂载

真实文件系统的一个接力，在很多嵌入式系统中，由于不需要外接大存储设备，所以initrd会作为永久的文件系统直接使用。

1.6 获得帮助

1.6.1 使用man page

目前Linux下有约2600个命令，每个命令的参数各异，所以不知道如何使用命令是很正常的。后面的章节中将会进一步学习基本的命令，不过仍然无法穷举所有命令的使用方法，那该怎么办呢？难道需要背下每一条命令吗？这条路自然是行不通的。幸运的是，由于所有的命令都属于自由软件，开发人员在开发这些命令时就考虑到这点，为了让使用者能够迅速地了解命令的用法，都会写出相关的说明文档，这就是man文件。不知道命令ls的使用方法吗？输入`man ls`，就会有一大堆说明告诉你怎么使用了。在查看man文件的时候，可以使用上下方向键阅读文件内容，也可以按空格键翻页，还可以使用关键字来搜索。比如说在`man ls`的页面上，输入“/time”，按回车键，就可以看到关键字被标记了。可以按小写字母n向下查找，也可以按大写的N向上查找，按小写字母q可以结束查看man文件。

我们在日常生活中，习惯于将不同的东西分门别类地存放，比如说在上学的时候会习惯性地吧数学类的辅导书放在一起，英语类的辅导书放在一起，这样可以方便寻找。同样，在Linux下也有这样的习惯，其中规定了以下9个man文件的种类：

- 常见命令的说明
- 可调用的系统
- 函数库
- 设备文件

- 文件格式
- 游戏说明
- 杂项
- 系统管理员可用的命令
- 与内核相关的说明

有些命令会在好几个种类中存在，可以使用`man-f`来查询要找的命令存在于哪些`man`文件中。例如：

```
[root@localhost ~]# man -f reboot
reboot          (2)  - reboot or enable/disable Ctrl-Alt-
Del
reboot [halt]   (8)  - stop the system
```

然后可用`man 2 reboot`或者`man 8 reboot`来分别查看`reboot`命令在`man`文件的第二章和第八章中的解释。

1.6.2 使用info page

info工具是一个基于菜单的超文本系统，包括少许关于Linux Shell、工具、命令的说明文档。比如可以在命令行中输入info ls来显示ls命令的说明文档：

```
[root@localhost ~]# info ls
File: coreutils.info, Node: ls invocation, Next: dir invoca
10.1 `ls': List directory contents
=====
The `ls' program lists information about files (of any type,
directories). Options and file arguments can be intermixed
arbitrarily, as usual.
.....(
略去内容).....
```

可以按空格键向下翻页，按PageUp、PageDown键上下翻页，按q键退出info查询。

1.6.3 其他获得帮助的方式

在学习Linux的过程中，也可以通过阅读红帽（RedHat）官方文档获得帮助，这些可以在互联网上轻易地找到。另外，一定要多利用互联网搜索引擎进行搜索，这对提高问题的排查能力是非常有帮助的。

在/usr/share/doc中，也有大量的帮助和说明文档，可以供日常查询参考。

第2章 Linux用户管理

2.1 Linux用户和用户组

Linux是一个多用户分时系统，想要使用系统资源，就必须在系统中有合法的账号，每个账号都有一个唯一的用户名，同时必须设置密码。在系统中使用用户的概念，一方面可以方便识别不同的用户，另一方面也可以为用户设置合理的文件权限，为每个用户的数据提供安全保障。另外，为了更灵活地管理用户和控制文件权限，Linux还采用了用户组的概念，这为系统管理提供了极大的便利。本节将具体介绍用户和用户组相关内容。

2.1.1 UID和GID

Linux系统如何区别不同的用户呢？可以很自然地想到，使用不同的用户名应该是一个好主意，就像真实世界中每个人都有名字一样。但“用户名”只是一种方便让人读的字符串，对机器来说是没有意义的。事实上，Linux系统采用一个32位的整数记录和区分不同的用户，这意味着系统可以记录多达40亿个不同的用户。这个用来区分不同用户的数字被称为User ID，简称UID。系统会自动记录“用户名”和UID的对应关系。Linux系统中的用户分为3类，即普通用户、根用户、系统用户。

普通用户是指所有使用Linux系统的真实用户，这类用户可以使用用户名及密码登录系统。Linux有着极为详细的权限设置，所以一般来说普通用户只能在其家目录、系统临时目录或其他经过授权的目录中操作，以及操作属于该用户的文件。通常普通用户的UID大于500，因为在添加普通用户时，系统默认用户ID从500开始编号。

根用户也就是root用户，它的ID是0，也被称为超级用户，root账户拥有对系统的完全控制权：可以修改、删除任何文件，运行任何命令。所以root用户也是系统里面最具危险性的用户，root用户甚至可以在系统正常运行时删除所有文件系统，造成无法挽回的灾难。所以一般情况下，使用root用户登录系统时需要十分小心。

系统用户是指系统运行时必须有的用户，但并不是指真实的使用者。比如在RedHat或CentOS下运行网站服务时，需要使用系统用户apache来运行httpd进程，而运行MySQL数据库服务时，需要使用系统用户mysql来运行mysqld进程。在RedHat或CentOS下，系统用户的ID范围是1~499。下面给出的示例显示的是目前系统运行的进程，第一列是运行该进程的用户。

```
[root@localhost ~]# ps aux
USER          PID %CPU %MEM    VSZ   RSS TTY      STAT START   T
root           1   0.0   0.0   2072   632 ?        Ss   Oct18   0
.....(
略去内容).....
apache       7930   0.0   0.1   9944   2064 ?        S    21:23   0:
```

在Linux系统中除了有用户之外，还有“用户组”的概念，不同的用户组同样也是用数字来区分的，这种用于区分不同用户组的ID被称为Group ID，也就是GID。

在下面的例子中，使用ls-l查看文件时，第三列和第四列显示的是这个文件的所有者是用户root，所有组是root组，如果加上了-n参数，第三列和第四列则是用UID和GID来显示的，这里分别是0和0。

```
[root@localhost ~]# ls -l anaconda-ks.cfg
-rw----- 1 root root 954 Oct  7 21:02 anaconda-ks.cfg
[root@localhost ~]# ls -ln anaconda-ks.cfg
-rw----- 1 0 0 954 Oct  7 21:02 anaconda-ks.cfg
```

那么，UID和GID又有什么联系呢？事实上，在Linux下每个用户都至少属于一个组。举个例子：每个学生在学校使用学号来作为标识，而每个学生又都属于某一个班级，这里的学号就相当于UID，而班级就相当于GID。当然了，每个学生可能还会同时参加一些兴趣班，而每个兴趣班也是不同的组。也就是说，每个学生至少属于一个组，也可以同时属于多个组。在Linux下也是一样的道理。既然是这样，如何查看自己的UID和GID呢？

要确认自己的UID，可以使用以下id命令来获得：

```
[root@localhost ~]# id
```



```
uid=0(root)gid=0(root)groups=0(root),1(bin),2(daemon),3(sys),
```

要确认自己所属的用户组，可以使用以下groups命令来获得：

```
[root@localhost ~]# groups
root bin daemon sys adm disk wheel
```

如果要查询当前在线用户，可在用户登录以后，使用命令who看到目前登录在系统中的所有用户。下面的例子说明当前有3个登录，其中用户root分别从tty1、pts/0登录到系统，而用户john从pts/1登录。

```
[root@localhost ~]# who
root      tty1          2012-10-22 00:13
root      pts/0         2012-10-22 21:20 (192.168.179.1)
john      pts/1         2012-10-22 22:35 (192.168.179.1)
```

2.1.2 /etc/passwd和/etc/shadow

前面已经说明，在登录Linux时必须输入用户名和密码。而系统用来记录用户名、密码最重要的两个文件就是/etc/passwd和/etc/shadow。以下是/etc/passwd中的几行内容：

```
[root@localhost ~]# cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
.....
（略去内容）.....
```

可以看到，虽然每行的内容不一样，但格式却是一致的，即每行都是使用6个分隔号“：”隔开的7列字符串。每一列所代表的含义如表2-1所示。

表2-1 etc/passwd内容格式说明

列数	含 义	说 明
1	用户名	是 UID 的字符串标记方式，方便阅读
2	密码	在旧的 UNIX 系统中，该字段是用户加密后的密码，现在已经不再使用，而是将密码放在 /etc/shadow 中，所以此处都只是一个字母 x
3	UID	系统用来区分不同用户的整数
4	GID	系统用来区分不同用户组的整数
5	说明栏	类似于“注释”，现在已经不使用
6	家目录	用户登录后，所处的目录，即用户家目录
7	登录 Shell	用户登录后，所使用的 Shell

从表2-1中可以了解到，/etc/passwd的第二列最早是在UNIX系统中用于记录密码的，但是这其中存在一个问题：由于每个用户都需要有读取这个文件的权限，而随着现代密码破解技术的发展，即便是加密的密码，也有被破解的可能，所以将密码从这个文件中剥离出去是非常必要的。

目前Linux的做法是，将密码相关的信息保存到/etc/shadow中，而且默认只有root用户才有读的权限，其他人完全没有读取这个文件的可能。这种密码保存方式被称为“影子密码”。看一下/etc/shadow中的第一行内容：

```
[root@localhost ~]# cat /etc/shadow
root:$1$JjIvgikC$YjiVyo3wVahvrwr0IETTV/:15620:0:99999:7:::
.....
（略去内容）.....
```

与/etc/passwd类似，/etc/shadow也是由冒号“:”隔开的，不同的是这里是8个冒号隔开的9列。每一列代表的含义如表2-2

所示。

表2-2 /etc/shadow内容格式说明

列数	含 义	说 明
1	用户名	是 UID 的字符串标记方式，方便阅读
2	密码	经过加密后的密码
3	密码的最近修改日	这个数字是从 1970 年 1 月 1 日至密码修改日的天数
4	密码不可修改的天数	修改密码之后，几天内不可修改密码，如果是 0，则随时可以修改
5	密码重新修改的天数	考虑到密码使用一段时间后可能会泄漏，可以设置一个修改时间，在密码到期之前系统会提醒用户修改密码
6	密码失效前提前警告的天数	设定密码到期前几天内开始提醒用户修改密码
7	密码失效宽限天数	如果密码到期，过了几天后将会失效，无法登录系统
8	账号失效日期	一般为空
9	保留字段	暂时没有使用

2.2 Linux账号管理

Linux账号管理是Linux系统管理员的一个重要工作，具体来说，涉及账号的添加、删除和修改等操作。从账号类型来说，Linux用户按照使用方式分为三种：一是根用户，二是系统用户，三是普通用户。

2.2.1 新增和删除用户

1.新增用户：useradd

`useradd`命令用于添加新的用户。其使用方式很简单，通常情况下可直接在该命令后跟上新增的用户名。比如，需要新建一个叫john的用户，直接输入命令`useradd john`即可。但是对于系统来说，完成这个命令需要在后台执行很多对用户来说毫无感知的行为。

```
[root@localhost ~]# useradd john
```

首先，系统需要将用户信息记录在`/etc/passwd`中，一般会在`/etc/passwd`和`/etc/shadow`末尾追加一条记录，同时会分配给该用户一个UID。

接着，要为该用户自动创建家目录。家目录以创建的用户名为目录名，创建的路径在`/home`目录中。比如，在上述案例中，创建的目录将是`/home/john`。

然后，复制`/etc/skel`下所有的文件至`/home/john`。说明一下，如果你使用`ls-l/etc/skel`命令查看，可以发现这个目录下“什么都没有”，但事实上，该目录下面有很多隐藏文件，使用命令`ls-la/etc/skel`就可以看到其中还是有好几个文件的。

最后，新建一个与该用户名一样的用户组，也就是说当创建用户john的时候，也同时创建了一个叫john的用户组，而用户john默认属于john用户组（关于用户组的概念将在下一节中讲到）。

这里需要对`/etc/skel`目录做一些说明。系统在添加用户时，

需要预先为这个用户创建一些默认的“配置文件”，而默认配置的就是/etc/skel目录下的几个隐藏文件。可以说，/etc/skel实际上是创建用户时的“模板”。

做一个小实验，实验过程如下所示：

```
[root@localhost ~]# cd /etc/skel/
[root@localhost skel]# touch TempFile
[root@localhost skel]# useradd john01
[root@localhost skel]# cd /home/john01/
[root@localhost john]# ls -l
total 0
-rw-r--r-- 1 john john 0 Oct 25 23:41 TempFile
```

也就是说，手工在/etc/skel中创建一个文件TempFile（touch命令就是创建文件的命令），然后再添加用户john01时，在家目录/home/john01中也会同样发现这个文件。这说明其实在创建用户后，会将/etc/skel中的所有文件直接复制过来。

在使用useradd添加用户时，系统会给该用户自动分配一个UID，但是也可以通过使用-u参数实现指定UID，当然，必须要指定的UID不与其他用户冲突才可以。下面的例子就创建了一个UID为555的账号：

```
[root@localhost skel]# useradd -u 555 user1
```

既然可以指定新创建用户的UID，也应该可以指定GID吧？答案是肯定的，使用-g参数可以做到这点。下面就是创建用户user2时，指定了该用户所属的Group是user1。

```
[root@localhost skel]# useradd -g user1 user2
```

还可以使用-d参数指定该用户的家目录，而不是使用系统默认创建的家目录。像下面这样就可以指定/home/mydir3作为user3用户的家目录：

```
[root@localhost skel]# useradd -d /home/mydir3 user3
```

useradd命令还有很多其他一些并不常用的参数，具体的参数和说明可以使用命令man useradd获得帮助。

2.修改密码：passwd

创建用户后，该用户实际上还没有登录系统的权限，因为在不设置密码的情况下，在/etc/shadow中该用户记录中以冒号分隔的第二列将显示为两个感叹号“！！”这说明不允许该用户登录系统。因此，需要同时设置用户的密码才行，设置命令是passwd后接用户名，如下所示：

```
[root@localhost skel]# passwd john
Changing password for user john.
New UNIX password:
Retype new UNIX password:
passwd: all authentication tokens updated successfully.
```

这里输入两遍需要设置的密码即可。如果不输入密码而是直接按回车键两次，密码会设置失败，如果输入了太过简单的密码，系统将会显示“BAD PASSWORD:it is too simplistic/systematic”。虽然系统会提示密码太过简单，但还是会接受其作为该用户的密码。

普通用户也可以使用passwd来修改自己的密码，但是需要提供当前用户的密码才可以，并且密码不能太过简单，因为系

统会拒绝普通用户设置过于简单的密码。命令如下所示：

```
[john@localhost ~]$ passwd
Changing password for user john.
Changing password for john
(current) UNIX password:
New UNIX password:
Retype new UNIX password:
passwd: all authentication tokens updated successfully.
```

与root用户使用这个命令的方式不同，普通用户在运行这个命令时，后面不能跟参数，哪怕是自己的用户名也不行。比如说使用john登录，然后采用passwd john命令，系统就会立刻报错，提示只有root用户才可以在后面跟上用户名，如下所示：

```
[john@localhost ~]$ passwd john
passwd: Only root can specify a user name.
```

3.修改用户：usermod

有时候可能会由于某些具体的场景，而需要对已存在的用户进行修改，这时候就需要使用usermod命令了。比如说，下面创建了一个用户并设置了密码：

```
[root@localhost ~]# useradd alice
[root@localhost ~]# passwd alice
Changing password for user alice.
New UNIX password:
Retype new UNIX password:
passwd: all authentication tokens updated successfully.
```

现在看一下用户alice在/etc/passwd中的记录：

```
[root@localhost ~]# cat /etc/passwd | grep alice
alice:x:503:503::/home/alice:/bin/bash
```

冒号隔开的第五列是用户alice的家目录，如果希望修改家目录为/home/alice_new，可使用以下命令对alice的家目录做修改：

```
[root@localhost ~]# usermod -d /home/alice_new -m alice
```

其中，-m参数的作用是，如果指定用户的家目录存在，就自动创建新目录/home/alice_new，并使用该目录作为alice的新家目录。如果没有这个参数，系统会报一个错误：
usermod:user/home/alice_new does not exist。再看一下alice在/etc/passwd中的记录，大家可以发现，第五列的家目录发生了变化：

```
[root@localhost ~]# cat /etc/passwd | grep alice
alice:x:503:503::/home/alice_new:/bin/bash
```

也许会因为某些原因，账号alice现在还不适合使用（如发现账号异常），需要暂时将这个账号冻结起来，这时，可以使用-L参数实现此目的。在操作之前先看一下/etc/shadow中关于alice的内容，然后再进行冻结操作，最后再看一下/etc/shadow，看看有什么不同。

```
[root@localhost ~]# cat /etc/shadow | grep alice
alice:$1$D0i70VUY$GmjQ6HijgNLsm7xnys4Lw/:15642:0:99999:7:::
[root@localhost ~]# usermod -L alice
[root@localhost ~]# cat /etc/shadow | grep alice
alice:!!$1$D0i70VUY$GmjQ6HijgNLsm7xnys4Lw/:15642:0:99999:7:::
```

你可能已注意到，在冒号隔开的第二列，也就是密码处多了一个感叹号，这表示该账号已被冻结。使用-U参数可以解锁，而且可以看到密码又恢复如从前了。

```
[root@localhost ~]# usermod -U alice
[root@localhost ~]# cat /etc/shadow | grep alice
alice:$1$Doi70VUY$GmjQ6HijgNLsm7xnys4Lw/:15642:0:99999:7:::
```

其实usermod命令就是在对/etc/passwd和/etc/shadow文件做一些修改而已。明白了这个道理之后，就算不使用这个命令依然可以手工对用户进行修改操作。usermod还有其他一些不常用的参数，具体的参数和说明可以使用man usermod命令获得帮助。

4.删除用户：userdel

用户管理除了创建、修改用户之外，有时候也需要删除用户。对应的命令是userdel。例如现在需要删除alice用户：

```
[root@localhost ~]# userdel alice
```

使用这个命令会将删除alice在/etc/passwd和/etc/shadow中的记录。但是从数据安全方面考虑，默认情况下，删除用户时并不会删除原来用户的家目录和邮件信息。可以使用-r参数同时删除用户家目录和该用户的邮件。注意，一旦执行了这条命令，该用户的相关文件就会被全部删除。

2.2.2 新增和删除用户组

1.增加用户组：groupadd

上一节中我们知道，在添加用户的时候系统默认会创建一个与用户名一样的用户组。其实也可以直接创建用户组，新增用户组的命令是groupadd，后接用户组名称作为其参数。在Linux中，使用/etc/group文件来记录用户组。如下所示为使用groupadd命令增加一个group1组：

```
[root@localhost ~]# groupadd group1
```

按回车键后，注意看/etc/group的最后一行，在本例中，添加的用户组group1的GID为503。

```
[root@localhost ~]# cat /etc/group
.....(
略去内容).....
group1:x:503:
```

在/etc/group文件中，每一行就代表一个用户组，其格式是使用3个分隔号“：”隔开的4列。第一列是用户组名，第二列代表密码（但是并不使用），第三列代表用户组的数字ID，第四列是组成员，这里为空说明还没有任何用户属于这个组。

2.删除用户组：groupdel

删除用户组的命令是groupdel，后接要被删除的用户组名作为其参数。这里需要注意的是，如果已有用户属于这个试图删除的组，该操作会失败。groupdel命令的使用方式如下：

```
[root@localhost ~]# groupdel group1
```

2.2.3 检查用户信息

1.查看用户：users、who、w

使用命令users可以查看当前系统有哪些用户。比如，在当前的系统中运行users命令，就会发现有二个root在当前机器上登录。实际上，Linux会把所有来自不同终端的活动定义为一个会话，从who命令的输出，可以看出root用户是通过不同的终端登录到系统中的。users命令相对比较简单，所以列出的信息也比较少，可以使用命令who来看到更多信息，如下所示：

```
[root@localhost ~]# users
root root
[root@localhost ~]# who
root      tty1          2012-11-01 23:00
root      pts/0         2012-11-01 22:56 (192.168.179.1)
```

命令显示的结果有3列，第一列是登录用户的用户名，第二列是用户登录的终端，第三列是用户登录的时间。如果是通过远程网络登录，则同时会显示远程主机的主机名或IP地址。还可以使用命令w看到更详细的信息，如下所示：

```
[root@localhost ~]# w
 23:21:30 up 27 min,  2 users,  load average: 0.00, 0.00, 0.0
USER      TTY      FROM          LOGIN@      IDLE        JCPU       PC
root      tty1      -             23:00       7.00s      0.02s      0.0
bash
root      pts/0     192.168.179.1 22:56       0.00s      0.03s      0.0
```

w命令的第一行会显示当前时间、系统运行时间、已登录的用户数量和系统负载。下面显示的信息分为8列，每一列解释如下。

第一列：登录用户的用户名。

第二列：用户登录终端。

第三列：如果用户从网络登录，则显示远程主机的主机名或IP地址。

第四列：用户登录时间。

第五列：用户闲置时间。

第六列：与终端相关的当前所有运行进程消耗的CPU时间总量。

第七列：当前WHAT列所对应的进程所消耗的CPU时间总量。

第八列：用户当前运行的进程。

2.调查用户：finger

`finger`命令在不加任何参数的情况下，同样会显示系统的登录用户，如下所示：

```
[root@localhost ~]# finger
```

Login	Name	Tty	Idle	Login Time	Office	Of
root	root	tty1	22	Nov 1 23:00		
root	root	pts/0		Nov 1 22:56	(192.168.179	

如果在`finger`后跟上某个用户名，则显示该用户更详细的信息，如下所示：

```
[root@localhost ~]# finger user1
```

Login: user1	Name: (null)
--------------	--------------

Directory: /home/user1

Shell: /bin/bash

Never logged in. #

显示用户最近一次登录到系统中的时间

No mail. #

显示邮件信息

No Plan. #

显示计划信息

2.3 切换用户

在使用Linux的过程中，很多时候由于实际需要可能要在不同的用户之间切换，比如，原本是使用普通用户登录的，但是在操作的过程中由于权限原因必须使用root用户来做一些操作，这时就需要临时切换到root用户；操作完成后，再退出切换到普通用户。其中将会涉及两种切换用户的方法：su和sudo，本节将针对这两种方法进行详细讲解。

2.3.1 切换成其他用户

在本章开头时就提到，Linux用户分为根用户（root）、普通用户、系统用户3种。其中根用户和普通用户是可以实际登录到系统中的。假如说我以普通用户john登录到系统中，这时候想使用useradd添加一个用户，怎么办？普通用户是没有添加用户的权限的，只有root用户才能创建用户。当然我们可以使用exit命令退出当前用户，然后使用root用户登录，再使用useradd添加用户。但是也有一种更方便的方式，那就是使用su命令，su是切换用户的意思。在不加参数的情况下，su命令默认表示切换到root用户，之后只要输入root密码就可以切换身份为root了，完成操作后，使用exit命令可以退出root切换到原先的用户。如下所示：

```
[john@localhost ~]$ su
Password:      #
输入root
用户的密码
[root@localhost john]# pwd
/home/john
[root@localhost john]# exit
exit
[john@localhost ~]$
```

su命令后面还可以加上一个“-”参数，就是键盘上的中横线。加上这个参数后，切换到root用户时，不但身份变成了root，而且还能应用root的用户环境。所谓“用户环境”就是/etc/passwd中定义的用户家目录、使用的Shell，以及关于这个用户的个性化设置等。如下所示：

```
[john@localhost ~]$ su -
Password:      #
输入root
```

```
用户的密码  
[root@localhost ~]# pwd  
/root  
[root@localhost ~]# exit  
logout  
[john@localhost ~]$
```

在演示su和su-这两个命令的时候，我在中间故意都运行了pwd命令，可以看到两次命令的显示是不一样的。真正的原因是使用su命令切换用户之后，当前的用户环境并没有发生变化，而使用su-命令切换用户后，用户环境变成root的了。

su-命令后还可以继续跟其他的用户名作为参数，这样就可以切换成指定用户的身份。比如说用户john在使用过程中需要临时切换成用户user1，这时就可以使用su-user1命令切换用户，但是同样需要知道user1的密码。值得一提的是，root用户可以使用su命令切换成任意用户而不需要密码，如下所示：

```
[john@localhost ~]$ su - user1  
Password: #  
输入用户user1  
的密码  
[user1@localhost ~]$
```

使用su命令虽然很方便，但还是有很明显的缺陷，就是切换成其他用户的前提是需要知道对方的密码。如果需要切换成root，那就需要root的密码。我们知道，root是系统中权限最高的用户，如果让太多人知道root密码，必然是很不安全的。为解决这个问题，我们可以使用sudo命令。

2.3.2 用其他用户的身份执行命令：sudo

上一节中了解了su命令，并且也知道了这个命令存在的缺陷。而sudo则通过一种可配置的方式解决了这个问题。该命令的使用方式是在sudo后跟上需要执行的命令，比如说sudo passwd user1，即使用root的身份修改user1的密码。运行该命令时，系统首先检查/etc/sudoers，判断该用户是否有执行sudo的权限，在确定有执行权限后，系统要求用户输自己的密码，如果密码输入正确，则会以root用户的身份运行passwd user1命令。

在演示sudo命令之前，首先需要设置/etc/sudoers这个配置文件。当然，可以使用一些常见的编辑器来编辑这个文件，比如vi或者vim编辑器等（常见编辑器的使用方法将在第9章中讲解），但是考虑到这个配置文件的重要性，Linux提供了专门编辑这个文件的方式，就是使用命令visudo来编辑这个文件，它的好处是可以在编辑后保存退出时自动检查语法设置，以防止不小心配置错误而造成无法使用sudo命令。该命令如下所示：

```
[root@localhost ~]# visudo
## Sudoers allows particular users to run various commands as
## the root user, without needing the root password.
##
## Examples are provided at the bottom of the file for collec
## of related commands, which can then be delegated out to pa
## users or groups.
##
## This file must be edited with the 'visudo' command.
.....(
略去内容).....
## Allow root to run any commands anywhere
root    ALL=(ALL)        ALL
john    ALL=(ALL)        ALL #
复制上一行的内容，并修改用户名为john
```

```
## Allows members of the 'sys' group to run networking, softw
## service management apps and more.
# %sys ALL = NETWORKING, SOFTWARE, SERVICES, STORAGE, DELEGAT
## Allows people in group wheel to run all commands
# %wheel      ALL=(ALL)      ALL
## Same thing without a password
# %wheel      ALL=(ALL)      NOPASSWD: ALL
## Allows members of the users group to mount and unmount the
## cdrom as root
# %users ALL=/sbin/mount /mnt/cdrom, /sbin/umount /mnt/cdrom
## Allows members of the users group to shutdown this system
# %users localhost=/sbin/shutdown -h now
```

修改完成后，使用用户john登录，然后再尝试使用sudo给别的用户修改密码，系统首先要求输入用户john的密码，验证通过后，就可以设置其他用户的密码了，如下所示：

```
[john@localhost ~]$ sudo passwd user1
[sudo] password for john: #
这里输入用户john
的密码
Changing password for user user1.
New UNIX password: #
输入user1
的新密码
Retype new UNIX password: #
再次输入user1
的新密码
passwd: all authentication tokens updated successfully.
```

加入的“john ALL=(ALL)ALL”这一行代表的意思是，john这个用户（第一列）可以从任何地方登录后（第二列的ALL）执行任何人（第三列的ALL）的任何命令（第四列的ALL）。还可以定义某一个组的sudo权限，比如“%john ALL=(ALL)ALL”可以让所有属于john用户组的用户从任何地方登录后执行任何人的任何命令。

正如上面例子所演示的，只需要知道自己的密码就可以使

用sudo执行任何命令，这样方便多了。但是每次都需要输入一遍密码也是比较麻烦的事情，想要实现不需要输入密码就可以执行命令，可以在最后一个ALL前添加“NOPASSWD:”，如下所示：

```
john    ALL=(ALL)        NOPASSWD:ALL
```

这样用户john在使用sudo时就不再需要输入密码了。实际上，将最后一个参数设置为ALL是很不安全的，因为这意味着用户实际拥有了全部的系统权限，和root的权限是一致的，在工作中可以根据用户实际的工作内容定义用户可以sudo执行的命令列表。假设用户john由于工作需要，经常要重启或者关闭服务器，那么就可以进行如下设置：

```
john    (ALL)          NOPASSWD:/sbin/shutdown, /usr/bin/reboot    ALL=
```

严格来说，sudo并不是真的切换了用户，而是使用其他用户的身份和权限执行了命令。

2.4 例行任务管理

日常生活中常会有很多例行性的事情，比如说每周工作日的闹钟、每年一次的生日提醒等。还有一些事情是偶发性的，比如突然需要处理一封紧急的邮件等。在Linux中也有处理这两种任务的方法。如果任务是周期性执行的，其命令为`cron`；如果只是在某一个特定的时间执行一次，其命令为`at`。

2.4.1 单一时刻执行一次任务：at

记得以前上网是需要用电话拨号的，不仅网速慢而且资费贵。有时候想要下载一些好玩的游戏软件，需要耗时很久，坐在那干等让人很着急。虽然随着技术和网速的发展，现在有很多下载工具都会在下载后自动断网或关机，但是当时并没有这些功能，于是我想了一个比较笨的办法，就是预估软件下载完成所需要的时间，然后在时间到了的时候自动关机。比如从现在开始，设置30分钟后自动关机，这时就可以使用at命令。

```
[root@localhost ~]# at now + 30 minutes
at> /sbin/shutdown -h now
at> <EOT>
job 1 at 2012-11-06 23:39
```

其中，第一行是定义从现在开始算，30分钟后安排一个任务；第二行是到了时间后执行关机操作；第三行是个<EOT>，这不是使用键盘输入的，而是使用了组合键Ctrl+D，表示输入结束；第四行是系统提示有一个任务将在23:39被执行。可以使用atq命令查看当前使用at命令调度的任务列表，第一列是任务编号；也可以使用atrm删除已经进入任务队列的任务，再使用atq查询时，发现已经没有任务列表了，如下所示：

```
[root@localhost ~]# atq
1          2012-11-06 23:39 a root #
查询at
的任务队列，第一个数字代表该任务的标号
[root@localhost ~]# atrm 1      #
删除标号为1
的任务
```

使用at还可以安排在具体的时间执行任务，比如说在午夜

12点实现自动关机，如下所示：

```
[root@localhost ~]# at 00:00 2012-11-07
at> /sbin/shutdown -h now
at> <EOT>
job 2 at 2012-11-07 00:00
[root@localhost ~]# atq
2          2012-11-07 00:00 a root
```

默认情况下，所有用户都可以使用at命令来调度自己的任务，如果由于特殊的原因需要禁止某些用户使用这个功能，可以将该用户的用户名添加至/etc/at.deny中。

2.4.2 周期性执行任务：cron

有一些任务是需要周期性执行的，比如说每天早晨的闹钟会在设定的时间准时响起。在Linux中，可以利用cron工具做这种设置。首先需要确定crond进程在运行，如果没有运行，需要先启动该进程。

```
[root@localhost ~]# service crond start
Starting crond: [ OK ]
[root@localhost ~]# service crond status
crond (pid 3257) is running...
```

用户可通过crontab来设置自己的计划任务，并使用-e参数来编辑任务。在这之前需要先了解一下设置的“语法”，当使用crontab-e进入编辑模式时，需要编辑执行的时间和执行的命令。在下面的示例中，前面5个*可以用来定义时间，第一个*表示分钟，可以使用的值是1~59，每分钟可以使用*和*/1表示；第二个*表示小时，可以使用的值是0~23；第三个*表示日期，可以使用的值是1~31；第四个*表示月份，可以使用的值是1~12；第五个*表示星期几，可以使用的值是0~6，0代表星期日；最后是执行的命令。当到了设定的时间时，系统就会自动执行定义好的命令，设置crontab的基本格式如下所示。

```
* * * * *      command
```

设置crontab的语法比较难以理解，这里举一些例子方便大家更好地理解，如下所示：

```
* * * * *      service httpd restart
*/1 * * * *    service httpd restart
```

```

#
这两种写法其实是一致的，都是每分钟重启httpd
进程。请注意，这只是一个例子，
除非你有确定的目的，否则不要在实际生产环境中这么设置
* */1 * * * service httpd restart
#
每小时重启httpd
进程
* 23-3/1 * * * service httpd restart
#
从23
点开始到3
点，每小时重启httpd
进程
30 23 * * * service httpd restart
#
每天晚上23
点30
分重启httpd
进程
30 23 1 * * service httpd restart
#
每月的第一天晚上23
点30
分重启httpd
进程
30 23 1 1 * service httpd restart
#
每年1
月1
日的晚上23
点30
分重启httpd
进程
30 23 * * 0 service httpd restart
#
每周日晚上23
点30
分重启httpd
进程

```

设置完成后，可以使用crontab-l查看设置的任务，也可以使用crontab-r删除所有的任务，如下所示：

```
[root@localhost ~]# crontab -l
30 23 * * 0 service httpd restart
[root@localhost ~]# crontab -r
[root@localhost ~]# crontab -l
no crontab for root
```

与at类似，每个用户都可以设置自己的crontab，如果由于特殊的原因需要禁止某些用户使用这个功能，可以将该用户的用户名添加至/etc/cron.deny中。除了root之外，普通用户只可以设置、查看、删除自己的计划任务，root可以使用-u参数查看指定用户的任务。比如root可以查看用户john的任务列表：

```
[root@localhost ~]# crontab -u john -l
```

2.4.3 /etc/crontab的管理

通过上一节，我们知道用户可以通过`crontab-e`命令来编辑定义自己的任务，事实上，系统也有自己的例行任务，而其配置文件是`/etc/crontab`。我们先来看一下这个文件的内容：

```
[root@localhost ~]# cat /etc/crontab
SHELL=/bin/bash
PATH=/sbin:/bin:/usr/sbin:/usr/bin
MAILTO=root
HOME=/
# run-parts
01 * * * * root run-parts /etc/cron.hourly
02 4 * * * root run-parts /etc/cron.daily
22 4 * * 0 root run-parts /etc/cron.weekly
42 4 1 * * root run-parts /etc/cron.monthly
```

英语基础比较好的人看到这个配置文件，都能猜出这个配置文件的意思，也就是定义了每小时、每天、每周、每月的任务。实际上`cron.hourly`、`cron.daily`、`cron.weekly`、`cron.monthly`都是文件夹，文件夹中则定义了具体的任务。

与使用`crontab-e`编辑的文件不同，“`#run-parts`”部分的第六列定义了以什么身份执行例行任务。这里的4个任务都是使用`root`来运行的。第七列定义了使用`run-parts`方式来运行第八列文件夹中的所有脚本。除了`run-parts`方式外，也可以使用命令模式运行例行任务，比如下面的例子就是定义了每分钟由`root`执行一次答应Hello的操作。

```
*/1 * * * * root echo "Hello"
```

第3章 Linux文件管理

3.1 文件和目录管理

几乎所有的计算机操作系统都使用目录结构组织文件。何为目录结构组织文件呢？具体来说就是在一个目录中存放子目录和文件，而在子目录中又会进一步存放子目录和文件，以此类推形成一个树形的文件结构，由于其结构很像一棵树的分支，所以该结构又被称为“目录树”。在Linux系统中也沿用了这种文件结构，所有目录和文件都在“根目录”下，目录名为“/”。FHS（文件系统层次标准）定义了根目录下的主要目录以及每个目录应该存放什么文件。下面进入根目录中，看一下Linux安装后默认的目录，如下所示：

```
[root@localhost ~]# cd /  
[root@localhost /]# ls  
bin dev home lost+found misc net proc sbin srv tmp var  
boot etc lib media mnt opt root selinux sys usr
```

根据FHS的定义，每个目录应该放置的文件如表3-1所示。

表3-1 FHS定义的目录结构

目录	目录的用途
/bin	常见的用户指令
/boot	内核和启动文件
/dev	设备文件
/etc	系统和服务的配置文件
/home	系统默认的普通用户的家目录
/lib	系统函数库目录
/lost+found	ext3 文件系统需要的目录，用于磁盘检查
/mnt	系统加载文件系统时常用的挂载点
/opt	第三方软件安装目录
/proc	虚拟文件系统
/root	root 用户的家目录
/sbin	存放系统管理命令

(续)

目录	目录的用途
/tmp	临时文件的存放目录
/usr	存放与用户直接相关的文件和目录
/media	系统用来挂载光驱等临时文件系统的挂载点

3.1.1 绝对路径和相对路径

1.绝对路径

正如前文所述，Linux系统采用了目录树的文件组织结构，在Linux下每个目录或文件都可以从根目录处开始寻找，比如：/usr/local/src目录。这种从根目录开始的全路径被称为“绝对路径”，绝对路径一定是以“/”开头的。

2.当前目录：pwd

想要确定当前所在的目录，可以使用以下pwd命令查看：

```
[root@localhost ~]# pwd
/root
```

3.特殊目录：（.）和（..）

在每个目录下，都会固定存在两个特殊目录，分别是一个点（.）和两个点（..）的目录。一个点（.）代表的是当前目录，两个点（..）代表的是当前目录的上层目录。在Linux下，所有以点开始的文件都是“隐藏文件”，对于这类文件，只使用命令ls-l是看不到的，必须要使用ls-la才可以看到，如下所示：

```
[root@localhost ~]# cd /mnt
[root@localhost mnt]# ls -la
total 16
drwxr-xr-x  2 root root 4096 Jan 27  2010 .
drwxr-xr-x 24 root root 4096 Jan  2 01:50 ..
```

4.相对路径

顾名思义，“相对路径”的关键在于当前在什么路径下。假设当前目录在/usr/local下，那么它的上层目录（/usr目录）可以用../表示，而/usr/local的下层目录（src）则可以用./src表示。前面讲到的（.）和（..）目录实际上也是属于相对路径，来看下面的例子：

```
[root@localhost ~]# cd /mnt    #
现在进入/mnt
目录
[root@localhost mnt]# ls -la
total 16
drwxr-xr-x  2 root root 4096 Jan 27  2010 .    #
代表当前目录
drwxr-xr-x 24 root root 4096 Jan  2 01:50 ..    #
代表上层目录
[root@localhost mnt]# cd .    #
进入当前目录（cd
命令后面再介绍）
[root@localhost mnt]# pwd    #
显示当前目录
/mnt    #
看到我们还是在/mnt
目录中
[root@localhost mnt]# cd ..    #
进入当前目录的上层目录
[root@localhost /]# pwd
/    #
进入了上层目录，也就是/
目录中
```

3.1.2 文件的相关操作

Linux遵循一切皆文件的规则，对Linux进行配置时，在很大程度上就是处理文件的过程，所以掌握文件的相关操作是非常有必要的。本节将介绍如何创建、删除、移动、重命名、查看文件，以及不同系统之间进行格式转换。后面的章节在具体介绍Linux编辑器的时候将会讲到如何编辑文件。

1.创建文件：touch

在Linux中创建一个文件，只需要进入相关目录，然后使用touch命令即可，参数为想要创建文件的文件名。比如说，在/tmp目录中创建一个test.txt文件：

```
[root@localhost ~]# cd /tmp
[root@localhost tmp]# touch test.txt    #
创建test.txt
[root@localhost tmp]# ls
-l    #
当前目录确实有了test.txt
total 0
-rw-r--r-- 1 root root 0 Jan  2 05:54 test.txt
```

事实上，如果在使用touch命令创建文件的时候，当前目录中已经存在了这个文件，那么这个命令不会对当前的同名文件造成影响，因为它并不会修改文件的内容，虽然实际上touch确实对该文件做了“修改”——它会更新文件的创建时间属性。比如说，在当前目录下我们继续使用touch test.txt命令，然后观察该文件时间属性部分的变化：

```
[root@localhost tmp]# touch test.txt    #
再次执行touch
命令
```

```
[root@localhost tmp]# ls -l
total 0
-rw-r--r-- 1 root root 0 Jan  2 06:03 test.txt  #
注意时间变化了
```

就会发现创建时间已经被修改了。

2.删除文件：rm

该命令是remove的简写，意思是“移除”。后面的参数是想要删除的文件的文件名，按回车键后系统会询问是否确认删除，这时候输入“y”然后按回车键即可。这里“y”的含义是yes，如果你现在反悔了，输入“n”后按回车键，将不会删除这个文件。

```
[root@localhost tmp]# rm test.txt
rm: remove regular empty file `test.txt'? y
```

3.移动或重命名文件：mv

该命令是move的简写，意思是“移动”。后面需要跟两个参数，第一个参数是要被移动的文件，第二个参数是移动到的目录。以下用一个示例来演示该命令的用法：

```
[root@localhost ~]# cd /tmp/  #
进入/tmp
目录
[root@localhost tmp]# ls  #
看一下目录中有什么
[root@localhost tmp]#  #
确认什么都没有
[root@localhost tmp]# touch test.txt  #
创建一个文件
[root@localhost tmp]# ls  #
再看一下目录内容
```

```
test.txt    #
确认文件已经创建成功了
[root@localhost tmp]# ls /mnt    #
看一下/mnt
目录中有什么
[root@localhost tmp]#          #
什么都没有
[root@localhost tmp]# mv test.txt /mnt/    #
移动文件到/mnt
下
[root@localhost tmp]# ls /mnt    #
再看一下/mnt
中有什么
test.txt    #
文件移动到/mnt
中了
[root@localhost tmp]# ls        #
看一下当前目录中的内容
[root@localhost tmp]#          #
文件已经被移走了
```

除了能移动文件，该命令还能重命名文件。接上例继续演示重命名文件的用法：

```
[root@localhost ~]# cd /mnt/    #
进入/mnt
目录
[root@localhost tmp]# ls        #
看一下当前目录中有什么
test.txt    #
这是刚刚移动过来的文件
[root@localhost mnt]# mv test.txt test.doc    #
修改了文件名
[root@localhost mnt]# ls
test.doc    #
确认文件名修改成功
```

上面两个例子分别演示了如何移动文件和重命名文件。其实mv还可以在移动文件的同时重命名文件。接着上例继续演示，如下所示：

```
[root@localhost mnt]# mv test.doc /tmp/test.txt
#
将test.doc
移动到/tmp
目录下，同时重命名为test.txt
[root@localhost mnt]# ls /tmp/
test.txt    #
检查一下，已重命名
```

这里需要注意的是，Linux下的目录也是一种“文件”，所以本节中所讲解的mv命令也同样适用于对目录的操作。

4.查看文件：cat

该命令是concatenate的简写，用户查看文件内容，后面跟上要查看的文件名即可。

```
[root@localhost ~]# cd    #
该命令将进入root
的家目录中
[root@localhost ~]# cat install.log    #
显示该文件内容
Installing setup-2.5.58-7.el5.noarch
warning:                                     setup-2.5.58-
7.el5: Header V3 DSA signature: NOKEY, key ID e8562897
Installing filesystem-2.4.0-3.el5.i386
Installing basesystem-8.0-5.1.1.el5.centos.noarch
.....(
略去内容).....
[root@localhost ~]# cat -n install.log    #
加上-n
参数可以显示每行的行号
    1  Installing setup-2.5.58-7.el5.noarch
        2          warning:      setup-2.5.58-
7.el5: Header V3 DSA signature: NOKEY, key ID e8562897
    3  Installing filesystem-2.4.0-3.el5.i386
    4  Installing basesystem-8.0-5.1.1.el5.centos.noarch
.....(
略去内容).....
```

5.查看文件头: head

有时候文件非常大,使用cat命令显示出来的内容太多,而我们可能并不想查看所有内容,只是想看看文件开始部分的内容,这时候就可以使用head命令了,后面跟上需要查看的文件名就可以了。默认情况下,head将显示该文件前10行的内容。继续拿install.log这个文件举例子,如下所示:

```
[root@localhost ~]# head install.log
Installing setup-2.5.58-7.el5.noarch
warning:                                     setup-2.5.58-
7.el5: Header V3 DSA signature: NOKEY, key ID e8562897
Installing filesystem-2.4.0-3.el5.i386
Installing basesystem-8.0-5.1.1.el5.centos.noarch
Installing tzdata-2010e-1.el5.noarch
Installing glibc-common-2.5-49.i386
Installing cracklib-dicts-2.8.9-3.3.i386
Installing nash-5.1.19.6-61.i386
Installing rmt-0.4b41-4.el5.i386
Installing centos-release-notes-5.5-0.i386
```

也可以使用-n参数指定显示的行数,如下所示:

```
[root@localhost ~]# head -n 20 install.log
Installing setup-2.5.58-7.el5.noarch
warning:                                     setup-2.5.58-
7.el5: Header V3 DSA signature: NOKEY, key ID e8562897
Installing filesystem-2.4.0-3.el5.i386
Installing basesystem-8.0-5.1.1.el5.centos.noarch
Installing tzdata-2010e-1.el5.noarch
Installing glibc-common-2.5-49.i386
Installing cracklib-dicts-2.8.9-3.3.i386
Installing nash-5.1.19.6-61.i386
Installing rmt-0.4b41-4.el5.i386
Installing centos-release-notes-5.5-0.i386
Installing 1:termcap-5.5-1.20060701.1.noarch
Installing mailcap-2.1.23-1.fc6.noarch
Installing dump-0.4b41-4.el5.i386
Installing gnu-efi-3.0c-1.1.i386
```



```
Installing rootfiles-8.1-1.1.1.noarch
Installing specspo-13-1.el5.centos.noarch
Installing man-pages-2.39-15.el5_4.noarch
Installing words-3.0-9.1.noarch
Installing libgcc-4.1.2-48.el5.i386
Installing glibc-2.5-49.i686
```

6.查看文件尾: tail

tail命令与head命令非常类似。当文件很大时，可以使用该命令查看文件尾部的内容，默认情况下tail也是只显示文件的最后10行内容，同样可以使用-n参数指定显示的行数。这里不具体举例子，大家参考上面head的用法就知道如何使用了。

但是tail还有个更实用的功能，就是可以动态地查看文件尾。这对查看一些不断改变的文件来说非常有用。比如说，系统中会有很多日志文件，这些文件是会随时变化的（具体地说，就是随时会有新的日志写入），要动态地查看这些文件，使用-f参数就可以做到。举个例子，/var/log/message文件是默认的系统日志文件，系统在运行中将会有大量的日志写入这个文件中，可以使用如下的命令，一旦有新的日志内容写入，该命令会立即将新内容显示出来。

```
[root@localhost~]#tail -f /var/log/messages
```

7.文件格式转换: dos2unix

该命令是DOS to UNIX的简写，也许你从字面上可以大概猜到它的作用，就是可以把DOS格式的文本文件转变成UNIX下的文本文件。之所以有这样的需求是因为Linux和Windows系统是通过文件共享的方式共享文件的，当把Windows下的文本文件移动到Linux下时，会由于系统之间文本文件的换行符不同而造成文件在Linux下的读写操作有问题。该命令的

使用方式非常简单直接，后面跟上需要转换的文件名即可。

3.1.3 目录的相关操作

与其他操作系统一样，Linux也有目录的概念，目录的作用在于存放其他的目录和文件。本节将介绍Linux下目录相关操作的方法。

1.进入目录：cd

该命令是change directory的简写，方便用户切换到不同的目录。以下是该命令使用方法的演示，示例中使用的pwd命令可以显示当前所处的目录：

```
[root@localhost ~]# cd #
无任何参数，默认进入root
的家目录
[root@localhost ~]# pwd #
看一下当前的目录
/root #
确实在root
的家目录中，也就是/root
[root@localhost ~]# cd /tmp #
进入/tmp
目录
[root@localhost tmp]# pwd
/tmp
[root@localhost tmp]# cd /mnt
[root@localhost mnt]# pwd
/mnt
```

2.创建目录：mkdir

该命令是make directory的简写，其用途是创建目录，使用方法是在后面跟上目录的名称，如下所示：

```
[root@localhost ~]# cd #
```

无任何参数，默认进入root的家目录

```
[root@localhost ~]# mkdir dir1      #
创建目录dir1
[root@localhost ~]# cd dir1/        #
进入目录dir1
[root@localhost dir1]# mkdir dir2 #
在dir1
中创建dir2
[root@localhost dir1]# cd dir2/    #
进入目录dir2
[root@localhost dir2]# pwd          #
显示当前目录
/root/dir1/dir2
```

如上面例子所示，创建一个目录后，又在该目录中创建了一个子目录。如果需要继续在dir2中创建dir3，然后在dir3中创建dir4，可以使用-p参数一次性创建所有目录，这样就不用费力地一个个创建了，命令如下所示：

```
[root@localhost dir2]# mkdir -p dir3/dir4
```

上面使用的是相对路径的方式，也可以使用绝对路径的方式来创建，如下所示：

```
[root@localhost dir2]# mkdir -p /root/dir1/dir2/dir3/dir4
```

大家可以任选一种来创建这种多层的目录结构。

3.删除目录：rmdir和rm

该命令是remove directory的简写，用来删除目录。但是需要注意的是，它只能删除空目录，如果目录不为空（存在文件或者子目录），那么该命令将拒绝删除指定的目录。继续上例，假设目前所在的目录是/root/dir1/dir2，当前目录下存在

dir3，但dir3不为空，因为包含dir4，我们试图使用该命令来删除dir3，结果如下：

```
[root@localhost dir2]# rmdir dir3/
rmdir: dir3/: Directory not empty
```

这里系统给出的信息是，dir3非空。如果先将dir3中的dir4目录删除，然后再删除dir3看看是否可行：

```
[root@localhost dir2]# rmdir dir3/dir4/  #
先删除dir4
目录
[root@localhost dir2]# rmdir dir3/      #
再删除dir3
[root@localhost dir2]#                  #
删除成功，系统无提示
```

由于dir4目录是空的，所以顺利删除了dir4，然后再删除dir3时就成功了。考虑一下这种情况：dir3中有成百上千个文件和目录，按照上面的方法，我们需要上千次地递归删除dir3下的所有文件和目录，直至dir3目录为空，然后才能删除该目录，这是否也太低效了？或者说是很愚蠢的方法。为了解决这个问题，可以使用rm命令来删除。

之前学习了如何使用rm来删除文件，如果需要使用它删除目录，只需要使用一个-r参数就可以做到，如下所示：

```
[root@localhost dir2]# cd
[root@localhost ~]#      #
进入/root
目录
[root@localhost ~]# rm -r dir1/
rm: descend into directory 'dir1/'? y  #
是否删除dir1
```

```
中的文件
rm: remove directory 'dir1//dir2'? y      #
是否删除dir2
目录
rm: remove directory 'dir1/'? y            #
是否删除dir1
目录
```

这样就删除了dir1目录，同时删除了目录中的所有其他目录。但是这里同样也存在一个问题：如果dir1中有数百个文件，那我们就需要不厌其烦地输入“y”来确认。命令rm在发现需要递归删除一个目录时，会尽量多地给你提示确认，希望以此引起管理员的注意，以加强操作的安全性，但是毕竟一次又一次地确认还是很烦琐的。所以，在使用rm删除目录时，最常用的组合参数是-rf，这样就不会有任何提醒了，可直接将目录删除干净。但是使用这个命令要极其小心，因为一旦删除了几乎是不可能恢复的了。另外，由于root用户在Linux系统中的权限非常高，甚至可以用rm-rf/命令来删除全部的系统文件（这样做的后果是灾难性的），所以使用-rf参数删除目录一定要慎之又慎！

4.文件和目录复制：cp

该命令是copy的简写，用于复制文件和目录。如果是复制文件，其后接两个参数，第一个参数是要复制的源文件，第二个参数是要复制到的目录或复制后的文件名，如下所示：

```
[root@localhost ~]#          #
现在是在/root
目录中
[root@localhost ~]# ls      #
查看一下该目录中有什么文件
anaconda-ks.cfg  install.log  install.log.syslog
[root@localhost ~]# cp anaconda-ks.cfg anaconda-ks-copy.cfg
```

如果是复制到其他目录中去，比如说复制到/tmp目录中，命令如下：

```
[root@localhost ~]# cp anaconda-ks.cfg /tmp/anaconda-ks-copy.cfg
```

如果想复制过去保持原文件名而不重命名，可以简单地写成下面的形式：

```
[root@localhost ~]# cp anaconda-ks.cfg /tmp/
```

复制目录同样也是使用cp命令。相对于复制文件，复制目录只需要使用-r参数即可，如下所示：

```
[root@localhost ~]# mkdir a      #  
创建a  
目录  
[root@localhost ~]# cp a b      #  
试图将a  
目录复制成b  
目录  
cp: omitting directory 'a'      #cp  
命令略过了目录，未发生复制  
[root@localhost ~]# cp -r a b   #  
加了-r  
参数后，复制成功
```

3.1.4 文件时间戳

之前在介绍touch命令的时候，已经知道通过touch可以创建新文件。如果文件已经存在，那么touch命令仅仅会更新文件的创建时间而不会修改文件内容。请记住，在Linux下目录也是一种文件，所以如果touch一个目录，这个目录的创建时间也会被更新。在下面的例子中，创建了一个文件touch_file1和一个目录touch_dir1，注意看一下时间是19:19，两分钟后，同时touch它们，再看一下时间就都变成了19:21。

```
[root@localhost ~]# mkdir touch_dir1
[root@localhost ~]# touch touch_file1
[root@localhost ~]# ll
drwxr-xr-x 2 root root 4096 Jan  3 19:19 touch_dir1
-rw-r--r-- 1 root root    0 Jan  3 19:19 touch_file1
#
两分钟后
[root@localhost ~]# touch touch_dir1 touch_file1
[root@localhost ~]# ll
drwxr-xr-x 2 root root 4096 Jan  3 19:21 touch_dir1
-rw-r--r-- 1 root root    0 Jan  3 19:21 touch_file1
```

不管是哪种系统，几乎所有的程序都会读写系统文件，默认情况下，一旦发生写文件操作，该文件的时间戳将会立刻得到更新。因此可以利用这种特性来有选择性地备份一些文件（又叫差异备份）。比如有一个目录中有若干个文件，我们每天需要备份一次。最简单的办法是每天使用cp操作全部备份一次，但是这种做法在文件总大小比较大的情况下会显得效率不高。如果有一些文件很大，但是和上一次备份相比并没有发生任何变化，实际上是不需要进行备份的，只需要找出在上一次备份之后发生变化的文件，然后备份这些文件即可。为了演示利用时间戳来进行差异化备份，下面先创建一些目录和文件：

```
[root@localhost ~]# mkdir org_dir      #
这是要备份的目录
[root@localhost ~]# mkdir bak_dir      #
这是备份存放目录
[root@localhost ~]# cd org_dir/        #
进入要备份的目录
[root@localhost org_dir]# touch a b c   #
创建几个文件
```

第一次备份自然是需要复制org_dir下的所有文件到bak_dir中：

```
[root@localhost org_dir]# cp * ../bak_dir/
#
复制当前目录下的所有文件到上层目录的bak_dir
目录中
#
这里用到了一个星号"*
", 代表所有, 还用到了相对目录
```

复制完成后, 在这个目录中利用touch命令创建出一个新文件time_stamp, 注意看一下, 该文件和其他文件的时间戳是不一样的, time_stamp文件的创建时间自然是比其他的文件要晚。下次备份的时候, 只需要找出比time_stamp文件时间戳新的文件, 然后备份该文件即可, 如下所示:

```
[root@localhost org_dir]# touch time_stamp
[root@localhost org_dir]# ll
total 0
-rw-r--r-- 1 root root 0 Jan  3 19:35 a
-rw-r--r-- 1 root root 0 Jan  3 19:35 b
-rw-r--r-- 1 root root 0 Jan  3 19:35 c
-rw-r--r-- 1 root root 0 Jan  3 19:43 time_stamp
```

假设再一次备份的时候, 文件a是被更新过了的, 这里使用touch来模拟一下这个场景: 使用touch命令发现a文件的时间

戳比time_stamp要新，那么可知a被程序修改过了，而其他的文件（文件b和文件c）并没有被更新，这时只需要备份a文件就可以了。备份完成后，还需要继续touch一下time_stamp，以更新该文件的时间戳，在下次备份的时候只需要找比这个文件时间戳更新的文件即可，如下所示：

```
[root@localhost org_dir]# touch a
[root@localhost org_dir]# ll
total 0
-rw-r--r-- 1 root root 0 Jan  3 19:47 a
-rw-r--r-- 1 root root 0 Jan  3 19:35 b
-rw-r--r-- 1 root root 0 Jan  3 19:35 c
-rw-r--r-- 1 root root 0 Jan  3 19:43 time_stamp
[root@localhost org_dir]# cp a ../bak_dir/
cp: overwrite '../bak_dir/a'? y
[root@localhost org_dir]# touch time_stamp
```

这里只是举一个例子来说明利用文件时间戳来做备份的原理，在实际工作中必须将这种过程脚本化、自动化。因为人为地备份文件一方面容易出错，另一方面也是不现实的：想象一下如果需要备份的文件有成百上千个，人工地逐个比较文件的时间戳是不可能的。

3.2 文件和目录的权限

可能大家早就有所耳闻，Linux系统之所以更安全，是因为对文件权限有着非常严格的控制。本节将要给大家介绍Linux系统中文件权限的概念，这些概念非常重要，了解和熟练掌握Linux下目录和文件的权限是必须的。但是，熟悉用户和用户组的概念是学习权限的前提，如果这方面存在疑问请仔细阅读第2章，否则很难理解本节的内容。

3.2.1 查看文件或目录的权限：ls-al

这已经不是我们第一次看到这个命令了，不过前面并没有详细介绍命令输出内容的含义，下面就来详细说明一下。其中，`-l`参数表示要求`ls`命令列出每个文件的详细信息，`-a`参数则要求`ls`命令还要同时列出隐藏文件。在`/root`目录中运行`ls-al`，然后看一下输出，如下所示：

```
[root@localhost ~]# ls -al
total 112
drwxr-x---  3 root root  4096 Oct  1 10:43 .
drwxr-xr-x 24 root root  4096 Oct  1 07:42 ..
-rw-----  1 root root  1017 Jan  2  2009 anaconda-ks.cfg
-rw-----  1 root root  5659 Sep 24 02:07 .bash_history
-rw-r--r--  1 root root    24 Jan  6  2007 .bash_logout
-rw-r--r--  1 root root   191 Jan  6  2007 .bash_profile
-rw-r--r--  1 root root   176 Jan  6  2007 .bashrc
-rw-r--r--  1 root root   100 Jan  6  2007 .cshrc
-rw-r--r--  1 root root 18590 Jan  2  2009 install.log
-rw-r--r--  1 root root    72 Oct  1 08:45 .lessht
drwx-----  2 root root  4096 Oct  1 08:48 .ssh
-rw-r--r--  1 root root   129 Jan  6  2007 .tcshrc
```

正如大家所见，`ls-al`格式化地输出了文件的详细信息，每个文件都有7列输出，下面详细介绍每列的含义。

第一列是文件类别和权限，这列由10个字符组成，第一个字符表明该文件的类型。表3-2列出了第一个字符可能的值和所代表的含义。接下来的属性中，每3个字符为一组，第2~4个字符代表该文件所有者（`user`）的权限，第5~7个字符代表给文件所有组（`group`）的权限，第8~10个字符代表其他用户（`others`）拥有的权限。每组都是`rw`x的组合，如果拥有读权限，则该组的第一个字符显示`r`，否则显示一个小横线；如果

拥有写权限，则该组的第二个字符显示w，否则显示一个小横线；如果拥有执行权限，则第三个字符显示x，否则显示一个小横线。

表3-2 字符含义

第一个字符可能的值	含 义
d	目录
-	普通文件
l	链接文件
b	块文件
c	字符文件
s	socket 文件
p	管道文件

第二列代表“连接数”，除了目录文件之外，其他所有文件的连接数都是1，目录文件的连接数是该目录中包含其他目录的总个数+2，也就是说，如果目录A中包含目录B和C，则目录A的连接数为4。

第三列代表该文件的所有人，第四列代表该文件的所有组，第五列是该文件的大小，第六列是该文件的创建时间或最近的修改时间，第七列是文件名。

3.2.2 文件隐藏属性

上一小节中介绍了文件权限属性，但这只是文件属性中的一部分。Linux下的文件还有一些隐藏属性，必须使用lsattr来显示，默认情况下，文件的隐藏属性都是没有设置的。查看文件的隐藏属性需要使用lsattr命令，如下所示：

```
[root@localhost ~]# lsattr anaconda-ks.cfg
----- anaconda-ks.cfg
```

结果中的第一列是13个小短横，其中每一个小横线都是一个属性，如果当前位置上设置了该属性就会显示相对应的字符。

如果要设置文件的隐藏属性，需要使用chattr命令。这里介绍几个常用的隐藏属性，第一种是a属性。拥有这种属性的文件只能在尾部增加数据而不能被删除。下面使用chattr来给该文件添加a属性：

```
[root@localhost ~]# chattr +a anaconda-ks.cfg
[root@localhost ~]# lsattr anaconda-ks.cfg
-----a----- anaconda-ks.cfg
[root@localhost ~]# rm anaconda-ks.cfg
rm: remove regular file 'anaconda-ks.cfg'? y
rm: cannot remove 'anaconda-ks.cfg': Operation not permitted
```

如上所示，设置了a属性的文件，即便是root用户也不能删除它，但是实际上可以以尾部新增（append）的方式继续向该文件中写入内容。

还有一种比较常用的属性是i属性。设置了这种属性的文件将无法写入、改名、删除，即便是root用户也不行。这种属性

常用于设置在系统或者关键服务中的配置文件，这对提升系统安全性有较大的帮助。

更多隐藏属性请使用`man chattr`查看。

3.2.3 改变文件权限：chmod

从前面内容可知，Linux下的每个文件都定义了文件拥有者（**user**）、拥有组（**group**）、其他人（**others**）的权限，我们使用字母**u**、**g**、**o**来分别代表拥有者、拥有组、其他人，而对应的具体权限则使用**rw****x**的组合来定义，增加权限使用**+**号，删除权限使用**-**号，详细权限使用**=**号。表3-3中用一些例子说明了如何使用**chmod**来改变文件的权限。

表3-3 chmod用例

作 用	命 令
给某文件添加用户读权限	chmod u+r somefile
给某文件删除用户读权限	chmod u-r somefile
给某文件添加用户写权限	chmod u+w somefile
给某文件删除用户写权限	chmod u-w somefile
给某文件添加用户执行权限	chmod u+x somefile
给某文件删除用户执行权限	chmod u-x somefile
添加用户对某文件的读写执行权限	chmod u+rw somefile
删除用户对某文件的读写执行权限	chmod u-rw somefile
给某文件设定用户拥有读写执行权限	chmod u=rwx somefile

如果要给用户组或其他人添加或删除相关权限，只需要将上面的**u**相应地更换成**g**或**o**即可。但是正如大家看到的，这种方式同一时刻只能给文件拥有者、文件拥有组或是其他所有人设置权限，如果要想同时设置所有人的权限就需要使用数字表示法了，我们定义**r=4**，**w=2**，**x=1**，如果权限是**rw****x**，则数字表示为**7**，如果权限是**r-x**，则数字表示为**5**。假设想设置一个

文件的权限是：拥有者的权限是读、写、执行（**rw**x），拥有组的权限是读、执行（**r**-x），其他人的权限是只读（**r**--），那么可以使用命令**chmod 754 somefile**来设置。

如果需要修改的不是一个文件而是一个目录，以及该目录下所有的文件、子目录、子目录下所有的文件和目录（即递归设置该目录下所有的文件和目录的权限），则需要使用**-R**参数，也就是**chmod-R 754 somedir**。

使用数字表示法设置权限是很常用的方式，读者一定要熟练掌握。

3.2.4 改变文件的拥有者：chown

该命令用来更改文件的拥有者，同时它也具备更改文件拥有组的功能。默认情况下，使用什么用户登录系统，那么该用户新创建的文件和目录的拥有者就是这个用户，比如使用root账户登录后，创建了一个文件a.txt，那么该文件的拥有者是root用户，如下所示：

```
[root@localhost ~]# touch a.txt
[root@localhost ~]# ll a.txt
-rw-r--r-- 1 root root 0 Jan  4 19:37 a.txt
```

要是想改变该文件的拥有者该怎么办呢？可使用chown命令将该文件的拥有者更改为john（假设系统中有这个用户）：

```
[root@localhost ~]# chown john a.txt
[root@localhost ~]# ls -l a.txt
-rw-r--r-- 1 john root 0 Jan  4 19:37 a.txt
```

该命令还可以同时更改文件的用户组。继续将该文件改为john用户组，使用方式如下：

```
[root@localhost ~]# chown :john a.txt
[root@localhost ~]# ls -l a.txt
-rw-r--r-- 1 john john 0 Jan  4 21:00 a.txt
```

以上两步可以使用一条命令同时设置：

```
[root@localhost ~]# chown john:john a.txt
```

如果需要修改的不是一个文件而是一个目录，以及该目录下所有的文件、子目录、子目录下所有的文件和目录（即递归设置该目录下所有的文件和目录的拥有者是john），则需要使用-R参数，也就是chown-R john somedir；如果要同时修改用户组为john，则使用chown-R john:john somedir。

3.2.5 改变文件的拥有组：chgrp

该命令用来更改文件的拥有组。下面将新创建的文件b.txt修改用户组为john：

```
[root@localhost ~]# touch b.txt
[root@localhost ~]# ls -l b.txt
-rw-r--r-- 1 root root 0 Jan  4 21:09 b.txt
[root@localhost ~]# chgrp john b.txt
[root@localhost ~]# ls -l b.txt
-rw-r--r-- 1 root john 0 Jan  4 21:10 b.txt
```

如果需要修改的不是一个文件而是一个目录，以及该目录下所有的文件、子目录、子目录下所有的文件和目录（即递归设置该目录下所有的文件和目录的拥有组是john），则需要使用-R参数，也就是chgrp-R john somedir。

3.2.6 文件特殊属性：SUID/SGID/Sticky

在介绍SUID之前，让我们来看一个奇怪的问题：通过前两章的学习我们已经了解到，每个用户都可以使用passwd（该命令的绝对路径是/usr/bin/passwd）来修改自己的密码。系统用于记录用户信息和密码的文件分别是/etc/passwd和/etc/shadow，命令passwd执行的最终结果是去修改/etc/shadow中对应用户的密码。对于这个文件，只有root用户有读权限，而普通用户在修改自己的密码时，最终也会修改这个文件。注意，虽然/etc/shadow文件对于root用户来说只有读权限，但是实际上root是可以使用强写的方式来更新这个文件的。但是普通用户在运行这个命令时居然有权限来写/etc/shadow文件，怎么可能呢？先来确认一下/etc/passwd和/etc/shadow的文件属性，从而确定普通用户根本没有写权限：

```
[root@localhost ~]# ls -l /etc/passwd
-rw-r--r-- 1 root root 1379 Dec 10 04:41 /etc/passwd
[root@localhost ~]# ls -l /etc/shadow
-r----- 1 root root 859 Dec 10 04:41 /etc/shadow
```

再来看一下/usr/bin/passwd的权限，发现有个特别的s权限在用户权限上，这就是奥秘所在——该命令是设置了SUID权限的，这意味着普通用户可以使用root的身份来执行这个命令。那么以上的疑问就很容易解释了。但是必须注意的是，SUID权限只能用于二进制文件。确认一下/usr/bin/passwd的权限：

```
[root@localhost ~]# ll /usr/bin/passwd
-rwsr-xr-x 1 root root 22984 Jan  7  2007 /usr/bin/passwd
```

下面是给一个二进制文件添加SUID权限的方法：

```
[root@localhost ~]# chmod u+s somefile
```

介绍完SUID之后，想必再来理解SGID就很容易了：如果某个二进制文件的用户组权限被设置了s权限，则该文件的用户组中所有的用户将都能以该文件的用户身份去运行这个命令，一般来说SGID命令在系统中用得很少，给一个二进制文件添加SGID权限的方法如下：

```
[root@localhost ~]# chmod g+s somefile
```

Sticky权限只能用于设置在目录上，设置了这种权限的目录，任何用户都可以在该目录中创建或修改文件，但是只有该文件的创建者和root可以删除自己的文件。RedHat或CentOS系统中的/tmp目录就拥有Sticky权限（注意看权限部分的最后是t），如下所示：

```
[root@localhost ~]# ll -d /tmp/  
drwxrwxrwt 3 root root 4096 Jan  4 04:53 /tmp/
```

举个例子，用户john登录后，在/tmp下创建了一个文件john_file，然后，用户jack也登录到系统中进入/tmp目录，他试图删除这个文件，系统会告诉他没有权限删除这个文件，如下所示：

```
#  
用户john  
登录到系统中并创建了/tmp/john_file  
[john@localhost ~]$ cd /tmp/
```

```
[john@localhost tmp]$ touch john_file
#
用户jack
登录到系统中试图删除/tmp/john_file
[jack@localhost ~]$ cd /tmp/
[jack@localhost tmp]$ rm john_file
rm: remove write-protected regular empty file 'john_file'? y
rm: cannot remove 'john_file': Operation not permitted
```

给一个目录添加t权限的方式如下：

```
[root@localhost ~]# chmod o+t somedir
```

3.2.7 默认权限和umask

既然说Linux系统对每个文件都有严格的权限控制，但是似乎到目前为止书中还并没有太细致地设置文件权限，而且在新创建文件的时候，也没有特意设置过权限。事实上，所有的文件在创建时就都是有权限的了，那么这些权限是怎么来的呢？也许你会想到是系统采用了默认权限的方法，也就是当我们创建文件的时候，系统套用默认权限来设置了文件。下面使用root用户登录系统来看一下：

```
[root@localhost tmp]# touch root_file1
[root@localhost tmp]# touch root_file2
[root@localhost tmp]# ls -l root_file1
-rw-r--r-- 1 root root 0 Jan  5 18:48 root_file1
[root@localhost tmp]# ls -l root_file2
-rw-r--r-- 1 root root 0 Jan  5 18:52 root_file2
[root@localhost tmp]# mkdir root_dir1
[root@localhost tmp]# mkdir root_dir2
[root@localhost tmp]# ls -ld root_dir1
drwxr-xr-x 2 root root 4096 Jan  5 18:54 root_dir1
[root@localhost tmp]# ls -ld root_dir2
drwxr-xr-x 2 root root 4096 Jan  5 18:54 root_dir2
```

注意，创建的root_file1、root_file2文件的权限都是644；创建的root_dir1、root_dir2目录的权限都是755。到这里似乎可以得出一个结论：文件的权限默认是644，目录的默认权限是755。但是实际情况是这样的吗？让我们使用普通用户john来操作一下，如下所示：

```
[john@localhost tmp]$ touch john_file1
[john@localhost tmp]$ touch john_file2
[john@localhost tmp]$ ls -l john_file1
-rw-rw-r-- 1 john john 0 Jan  5 19:00 john_file1
[john@localhost tmp]$ ls -l john_file2
-rw-rw-r-- 1 john john 0 Jan  5 19:00 john_file2
```



```
[john@localhost tmp]$ mkdir john_dir1
[john@localhost tmp]$ mkdir john_dir2
[john@localhost tmp]$ ls -ld john_dir1
drwxrwxr-x 2 john john 4096 Jan  5 19:00 john_dir1
[john@localhost tmp]$ ls -ld john_dir2
drwxrwxr-x 2 john john 4096 Jan  5 19:00 john_dir2
```

这里创建的john_file1、john_file2文件的权限都是664；创建的john_dir1、john_dir2目录的权限都是775。

可以给出一个结论：对于root用户，文件的默认权限是644，目录的默认权限是755；对于普通用户，文件的默认权限是664，目录的默认权限是775。到这里似乎可以结束关于默认权限的讨论了。但是，有两个疑问请读者考虑一下：

- 这个默认权限是从哪里来的呢？
- 为什么root用户和普通用户的默认权限不同呢？

要想回答上面的问题，就需要引入umask概念，中文翻译为：遮罩。在Linux下，定义目录创建的默认权限的值是“umask遮罩777后的权限”，定义文件创建的默认权限是“umask遮罩666后的权限”。

系统在/etc/profile文件中，通过第51行至55行的一段代码设置了不同用户的遮罩值。

```
if [ $UID -gt 99 ] && [ "`id -gn`" = "`id -un`" ]; then
    umask 002
else
    umask 022
fi
```

从上面的代码中可以看出，UID大于99的用户设置了umask为002，否则为022。所以umask值对于root用户是022，对于普

通用户是002，这也就造成了上面我们看到的root用户和普通用户创建出来的文件和目录默认权限不一样，那么如何使用遮罩计算权限呢？

777用字符串表示为：rwxrwxrwx，如果遮罩值是022，用字符串表示为：----w--w-，那么前者第五位和第八位的w被遮罩掉，权限变为rwxr-xr-x，用数字表示就是755。如果遮罩值是002，用字符串表示为：-----w-，那么第八位的w被遮罩掉，权限变为rwxrwxr-x，用数字表示就是775。

666用字符串表示为：rw-rw-rw-，如果遮罩值是022，用字符串表示为：----w--w-，那么前者第五位和第八位的w被遮罩掉，权限变为rw-r--r--，用数字表示就是644。如果遮罩值是002，用字符串表示为：-----w-，那么第八位的w被遮罩掉，权限变为rw-rw-r--，用数字表示就是664。

特别强调一下，网络上有很多关于计算umask遮罩后权限值的讲解，比较主流但是错误的讲解方式是使用“同位相减”的做法来计算遮罩后的值，比如说777-022同位相减得到755，666-022同位相减得到644，这种看似正确的结果其实只是一种巧合，并不是了解遮罩的正确方式。假设有个文件的权限为666，在遮罩值为011的情况下，采用该“同位相减”的方法计算出的权限值为655，但实际上正确的权限值应该是666。这点请读者注意。

3.2.8 查看文件类型：file

之前已经讲到，使用ls-l命令可以通过查看第一个字符判断文件类型。字母d代表目录、字母l代表连接文件，字母b代表块文件，字母c代表字符文件，字母s代表socket文件，字符-代表普通文件，字母p代表管道文件，而file命令则可以直接告诉我们文件类型，还能给出更多的文件信息，如下所示：

```
#!/root
是一个目录
[john@localhost ~]$ file /root
/root: directory
#!/tmp
是一个拥有sticky
属性的目录
[john@localhost ~]$ file /tmp
/tmp: sticky directory
#
使用ls
-l
命令查看，显示这是一个普通文件
[john@localhost ~]$ ls -l /etc/passwd
-rw-r--r-- 1 root root 1453 Jan  4 18:12 /etc/passwd
#
使用file
命令查看，显示这是一个ASCII
编码的文本文件
[john@localhost ~]$ file /etc/passwd
/etc/passwd: ASCII text
#
使用ls
-l
命令查看，显示这是一个普通文件，看不出与/etc/passwd
的差别
[john@localhost ~]$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 22984 Jan  7 2007 /usr/bin/passwd
#
使用file
命令查看，显示这是一个32
位的可执行性二进制文件
[john@localhost ~]$ file /usr/bin/passwd
```

/usr/bin/passwd: setuid ELF 32-
bit LSB executable, Intel 80386, version 1 (SYSV),
for GNU/Linux 2.6.9, dynamically linked (uses shared libs), f

3.3 查找文件

操作系统中有成千上万的文件散落在文件系统的各个角落中，还有不同用户创建的各种文件。随着系统的运行，文件数会越来越多，要想记住所有的文件在什么位置是不可能的。可能大家已经熟悉了在Windows下使用搜索工具来查找文件，但是在Linux系统下由于主要使用的是字符界面（图形界面上也没有类似的工具），那么查找文件就只能通过一些查找命令来进行。

3.3.1 一般查找：find

这个命令言简意赅地道出了其作用，不需要更多解释。在某个路径下查找文件的方法如下：

```
find PATH -name FILENAME
```

假设需要在系统中找到一个名为httpd.conf的文件，可以这么写：

```
[root@localhost ~]# find / -name httpd.conf
```

这条命令的意思是，从根目录开始寻找名为httpd.conf的文件。由于是从根目录开始寻找，find命令会遍历/下的所有文件，然后打印出寻找结果。如果你有点经验，大概知道这个文件可能存在于/etc下，因为看起来这是一个配置文件，这时便可以优化一下查找语句，这样耗时会更少一点。命令如下所示：

```
[root@localhost ~]# find /etc -name httpd.conf
```

可以使用星号通配符来模糊匹配要查找的文件名，比如想找出系统中所有以.conf结尾的文件，或以httpd开头的文件：

```
[root@localhost ~]# find / -name *.conf  
[root@localhost ~]# find / -name httpd*
```

其实find还有很多参数可以使用，如表3-4所示。更多用法

请使用man find来获得帮助。

表3-4 find常见参数

参 数	含 义
-name filename	查找文件名为 filename 的文件
-perm	根据文件权限查找
-user username	根据用户名查找
-mtime -n/+n	查找 n 天内 /n 天前更改过的文件
-atime -n/+n	查找 n 天内 /n 天前访问过的文件
-ctime -n/+n	查找 n 天内 /n 天前创建的文件
-newer filename	查找更改时间比 filename 新的文件
-type b/d/c/p/l/f/s	查找块 / 目录 / 字符 / 管道 / 链接 / 普通 / 套接字文件
-size	根据文件大小查找
-depth n	最大的查找目录深度

3.3.2 数据库查找: locate

与find不同，locate命令依赖于一个数据库文件，Linux系统默认每天会检索一下系统中的所有文件，然后将检索到的文件记录到数据库中。在运行locate命令的时候可直接到数据库中查找记录并打印到屏幕上，所以使用locate命令要比find命令反馈更为迅速。在执行这个命令之前一般需要执行updatedb命令（这不是必须的，因为系统每天会自动检索并更新数据库信息，但是有时候会因为文件发生了变化而系统还没有再次更新而无法找到实际上确实存在的文件。所以有时需要主动运行该命令，以创建最新的文件列表数据库），以及时更新数据库记录。下面是使用locate命令来查找httpd.conf文件：

```
[root@localhost ~]# updatedb
[root@localhost ~]# locate httpd.conf
/etc/httpd/conf/httpd.conf
```

为了让大家更好地理解locate的工作原理，在这里给大家展示一个实验，如下所示：

```
#
创建一个文件
[root@localhost ~]# touch test_locate
#
用find
命令查找
[root@localhost ~]# find / -name test_locate
/root/test_locate    #
找到了
#
再用locate
找一下
[root@localhost ~]# locate test_locate
[root@localhost ~]#    #
没找到！为什么？
```



```
#
执行一下updatedb
, 更新数据库
[root@localhost ~]# updatedb
[root@localhost ~]# locate test_locate
/root/test_locate    #
找到了! 说明由于没有更新数据库, 所以无法使用locate
命令找到刚创建的文件
#
将该文件删除
[root@localhost ~]# rm test_locate
rm: remove regular empty file 'test_locate'? y    #
确认删除了
#
再次locate
, 但仍然可以找到
[root@localhost ~]# locate test_locate
/root/test_locate
#
用updatedb
再次更新一下
[root@localhost ~]# updatedb
[root@localhost ~]# locate test_locate
[root@localhost ~]#      #
再找, 没有这个文件了
```

这个实验表明, locate命令依赖于其用于记录文件的数据库, 该数据库需要使用updatedb来更新。当然, 系统每天也会自动运行一次, 但是不必等系统运行, 必要的时候可主动进行手动更新。

3.3.3 查找执行文件：which/whereis

which用于从系统的PATH变量所定义的目录中查找可执行文件的绝对路径。比如说想查找passwd这个命令在系统中的绝对路径，使用方法如下：

```
[root@localhost ~]# which passwd
/usr/bin/passwd
```

使用**whereis**也能查到其路径，但是和**which**不同的是，它不但能找出其二进制文件，还能找出相关的man文件：

```
[root@localhost ~]# whereis passwd
passwd: /usr/bin/passwd /etc/passwd /usr/share/man/man5/passw
/usr/share/man/man1/passwd.1.gz
```

3.4 文件压缩和打包

记得在使用软盘的时期，我经常为需要复制的文件大于1.44MB而感到麻烦，因为那时候单张软盘最大的容量只有1.44MB。这时通常要将文件做一下压缩处理，运气好的话压缩后就可以复制到一张软盘上了，但是大多数情况下即便文件经过压缩还是很大，这时还得借用分卷工具将压缩过的文件分成若干个小文件，再使用多张软盘复制。随着科技的发展，现在存储设备的容量越来越大，虽然在日常生活中家用计算机已经不太需要通过压缩文件的方式来获得更多的空间，但是在服务器环境中，还是非常必要的。比如，有很多应用非常繁忙，每天的交易巨大，所产生的日志动辄上百GB，但是由于数据本身是有意义的，不能简单地删除，所以通过压缩的方式来存储日志是非常普遍的做法。且文本类的日志文件之压缩比往往能达到80%~90%，这样节省下来的空间是非常可观的。本节就来学习一下Linux下的常见压缩和解压缩工具以及它们的使用方法。

3.4.1 gzip/gunzip

gzip/gunzip是用来压缩和解压缩单个文件的工具，使用方法比较简单。比如，在/root目录下压缩install.log文件，压缩后生成的文件是install.log.gz文件，然后再使用gunzip文件将其解压缩即可。如下所示：

```
[root@localhost ~]# gzip install.log
[root@localhost ~]# ls install.log.gz
install.log.gz
[root@localhost ~]# gunzip install.log.gz
```

3.4.2 tar

tar不但可以打包文件，还可以将整个目录中的全部文件整合成一个包，整合包的同时还能使用gzip的功能进行压缩，比如说把整个/boot目录整合并压缩成一个文件。一般来说，整合后的包习惯使用.tar作为其后缀名，使用gzip压缩后的文件则使用.gz作为其后缀名。因为tar有同时整合和压缩的功能，所以可使用.tar.gz作为后缀名，或者简写为.tgz。下面的命令将/boot目录整合压缩成了boot.tgz文件：

```
[root@localhost ~]# tar -zcvf boot.tgz /boot
```

这里-z的含义是使用gzip压缩，-c是创建压缩文件（create），-v是显示当前被压缩的文件，-f是指使用文件名，也就是这里的boot.tgz文件。解压命令如下：

```
[root@localhost ~]# tar -zxvf boot.tgz
```

上面的命令会直接将boot.tgz在当前目录中解压成boot目录，-z是解压的意思。如需要指定压缩后的目录存放的位置，需要再使用-C参数。比如说将boot目录解压到/tmp目录中：

```
[root@localhost ~]# tar -zxvf boot.tgz -C /tmp
```

3.4.3 bzip2

使用bzip2压缩文件时，默认会产生以.bz2扩展名结尾的文件，这里使用-z参数进行压缩，使用-d参数进行解压缩。

```
[root@localhost ~]# bzip2 install.log
[root@localhost ~]# ls -l install.log.bz2
-rw-r--r-- 1 root root 3588 Dec 10 03:08 install.log.bz2
[root@localhost ~]# bzip2 -d install.log.bz2
```

3.4.4 cpio

该命令一般是不单独使用的，需要和find命令一同使用。当由find按照条件找出需要备份的文件列表后，可通过管道的方式传递给cpio进行备份，生成/tmp/conf.cpio文件，然后再将生成的/tmp/conf.cpio文件中包含的文件列表完全还原回去。

```
#
备份:
[root@localhost ~]# find /etc -name *.conf | cpio -
cov > /tmp/conf.cpio
#
还原:
[root@localhost ~]# cpio --absolute-filenames -
icvu < /tmp/conf.cpio
```

第4章 Linux文件系统

4.1 文件系统

通过前几章的学习，大家已经知道Linux使用了树形文件存储结构，在磁盘上存储文件的时候，使用的则是目录加文件的形式。但实际上对于磁盘等各种存储设备来说，无论是什么数据，都只有0和1的概念。磁盘的盘片是带磁性的，物理上最终会使用不同的磁性代替0和1，这里就有一个明显的问题：磁盘的物理存储方式决定了其根本没有文件和目录的概念。而对用户来说，0和1同样毫无意义，那怎么办呢？这就需要一种类似于“翻译”的机制存在于用户和磁盘之间了，在Linux中采用的是文件系统+虚拟文件系统（Virtual File System, VFS）的解决方案。

4.1.1 什么是文件系统

文件系统是操作系统用于明确磁盘或分区上相关文件的方法和数据结构，通俗的说法就是在磁盘上组织文件的方法。在使用前，都需要针对磁盘做初始化操作，并将记录的数据结构写到磁盘上，这种操作就是建立文件系统，在有些操作系统中称之为格式化。

Linux支持多种不同的文件系统，包括ext2、ext3、ext4、zfs、iso9660、vfat、msdos、smbfs、nfs等，还能通过加载其他模块的方式支持更多的文件系统。虽然文件系统多种多样，但是大部分Linux系统都具有类似的通用结构，包括超级块（superblock）、i节点（inode）、数据块（data block）、目录块（directory block）等。其中，超级块包括文件系统的总体信息，是文件系统的核心，所以在磁盘中会有多个超级块，以防止由于磁盘出现坏块导致全部文件系统无法使用。i节点存储所有与文件有关的元数据，也就是文件所有者、权限等属性数据以及指向的数据块，但是不包括文件名和文件内容。数据块是真实存放文件数据的部分，一个数据块默认情况下是4KB。目录块包括文件名和文件在目录中的位置，并包括文件的i节点信息。

4.1.2 ext2文件系统简介

Linux最早引入的文件系统类型是minix，由于其存在一定的局限性，比如说文件名最长仅支持14个字符，文件最大为64MB等因素，后来被ext2（The Second Extended File System）文件系统所取代，该文件系统有着极好的存储性能，所以曾一度成为Linux中的标准文件系统。和很多文件系统一样，ext2文件系统也是采取将文件数据存放到数据块中的方式来存储数据的，这些数据块的大小可以在创建文件系统的时候指定，对于存放的每个文件和目录，都会有一个inode指定，文件系统中所有的inode都是使用inode表来进行记录的，一定数量的块就会组成一个块组。在ext2文件系统中，整个分区的文件系统信息都被存放在超级块中，考虑到超级块所具有的重要性，因此在每个块组的开头中都有相同的备份。

但是ext2文件系统的弱点也是很明显的：它不支持日志功能。这很容易造成在一些情况下丢失数据，这个天然的弱点让ext2文件系统无法用于关键应用中，目前已经很少有企业使用ext2文件系统了。

4.1.3 ext3文件系统简介

为了弥补ext2文件系统的不足，有日志功能的ext3文件系统应运而生了。它直接从ext2文件系统发展而来，所以完全兼容ext2文件系统，而且支持从ext2非常简单地（只需要两条命令）转换为ext3，这种特性让也更多的老用户转而使用ext3文件系统。

那么为什么需要日志文件系统呢？因为日志文件系统使用了“两阶段提交”的方式来维护待处理的事务。比方说在写入数据之前，文件系统会先在日志中写入相关记录信息，然后再开始真实地写数据，写完数据后则会将之前写入日志中的内容删除。这样一来，如果遇到问题需要检查文件系统或对ext3文件系统进行修复时，只需要检查日志即可，而ext2修复文件系统时，则需要遍历整个文件系统来检查文件的一致性信息，因此ext3节省了大量修复文件系统所需要的时间。不过，由于增加了日志功能，在存取数据时ext3文件系统要比ext2所做的写入操作多，但是ext3对写操作做了优化，使其性能不会比ext2低。

4.2 磁盘分区、创建文件系统、挂载

磁盘使用前需要对其进行分割，这种动作被形象地称为分区。磁盘的分区分为两类，即主分区和扩展分区。受限制于磁盘的分区表大小（MBR大小为512字节，其中分区表占64字节），由于每个分区信息使用16字节，所以一块磁盘最多只能创建4个主分区，为了能支持更多分区，可以使用扩展分区（扩展分区中可以划分更多逻辑分区），但是即便这样，分区还是要受主分区+扩展分区最多不能超过4个的限制。在完成磁盘分区后，需要进行创建文件系统的操作，最后将该分区挂载到系统中的某个挂载点才可以使用。

4.2.1 创建文件系统：fdisk

下面将使用VMware虚拟机演示如何使用fdisk。首先在虚拟机设置中添加一块磁盘，方式如图4-1～图4-6所示。完成后启动虚拟机。

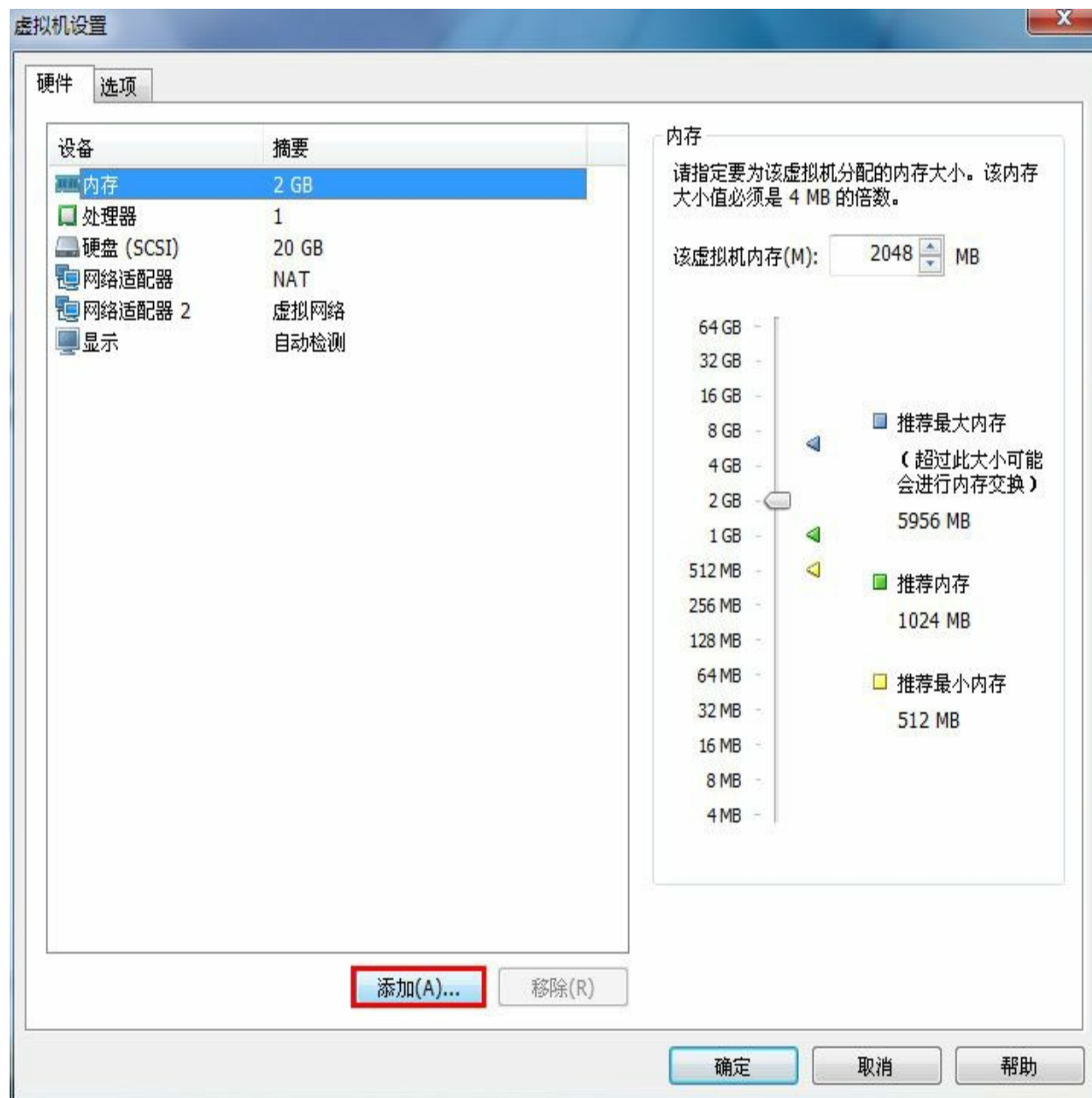


图4-1 选择“添加”，为虚拟机增加设备

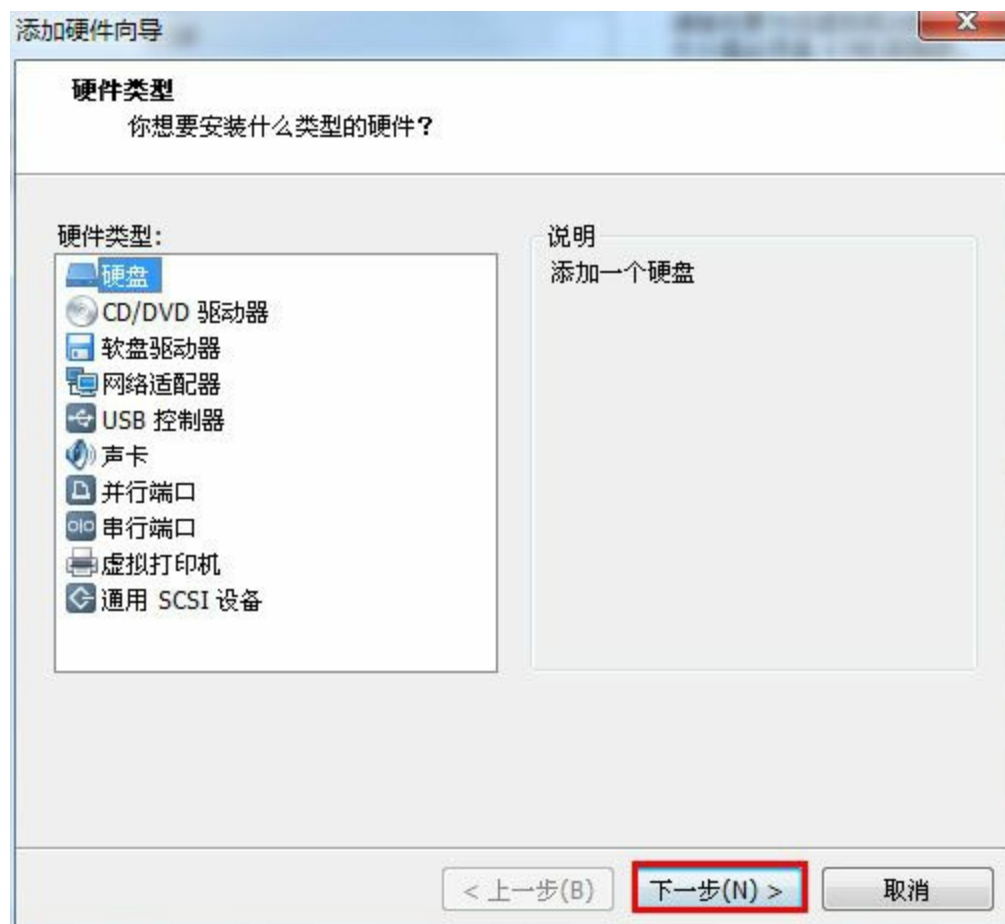


图4-2 选择“硬盘”



图4-3 选择“创建一个新的虚拟磁盘”



图4-4 选择SCSI选项

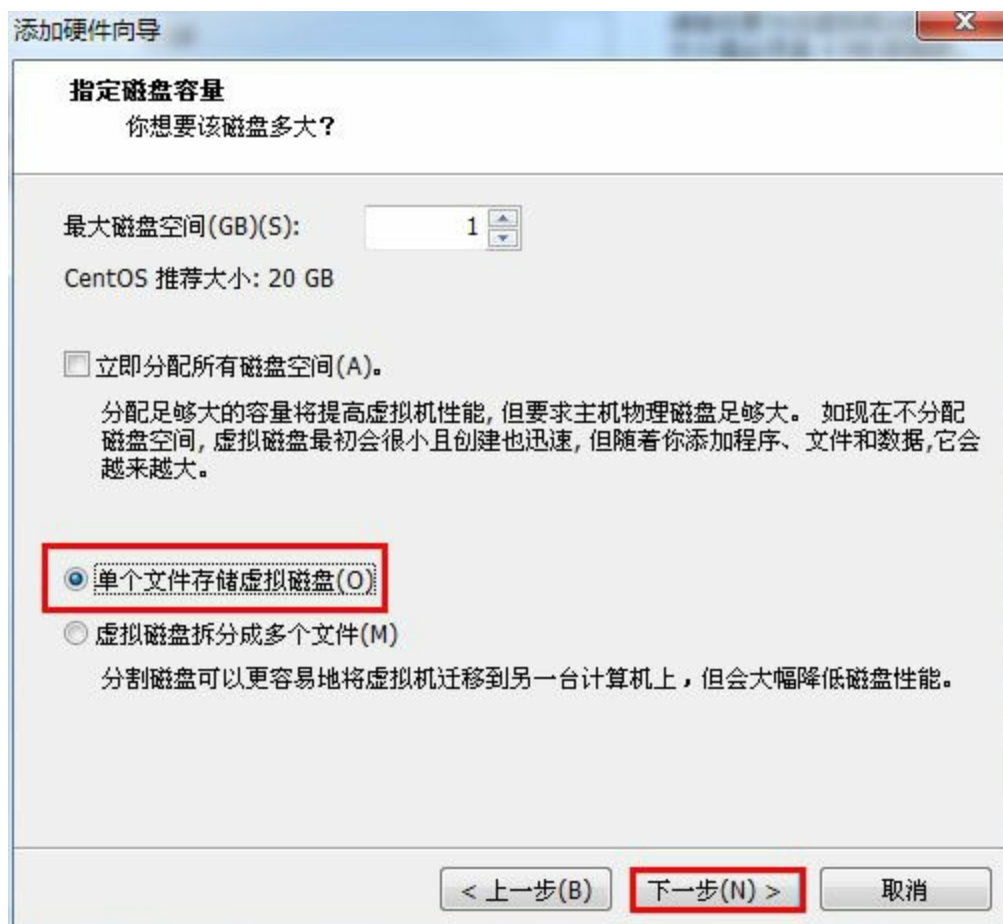


图4-5 选择“单个文件存储虚拟磁盘”选项

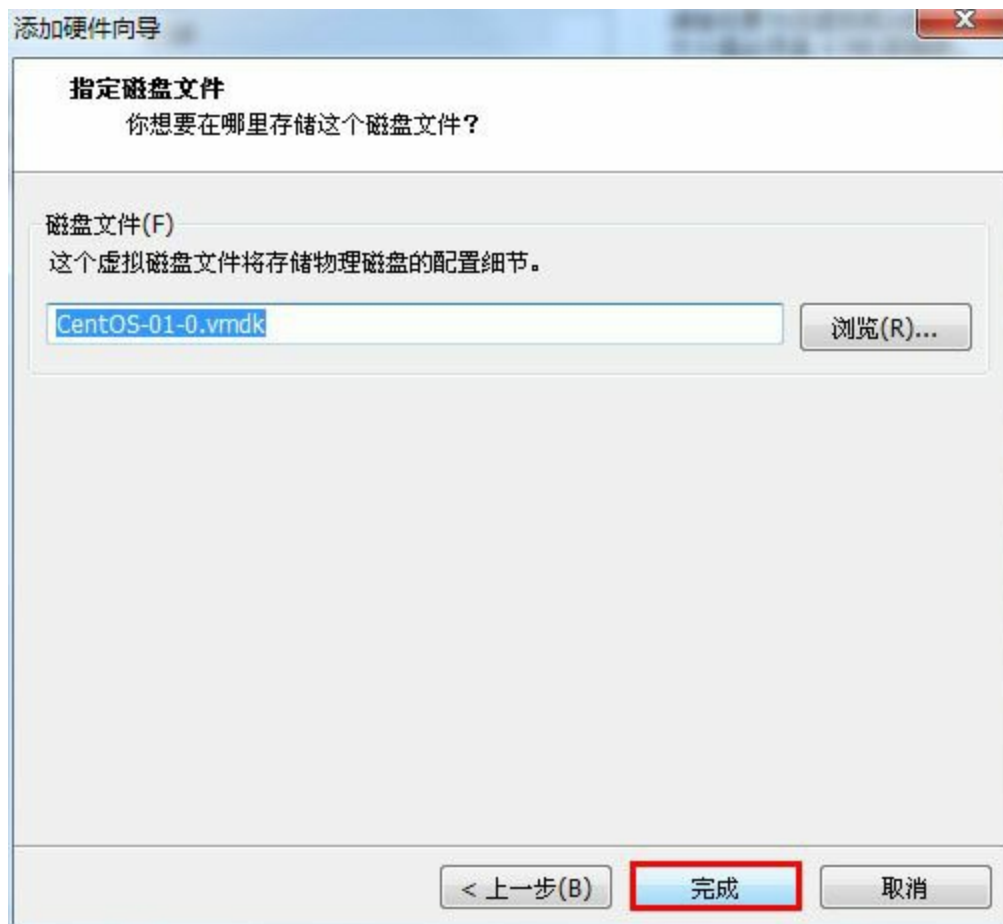


图4-6 命名磁盘

重新启动虚拟机后，使用fdisk-l查看一下发现，有一个/dev/sdb设备，这就是新添加的磁盘在操作系统中对应的设备文件。其大小是1073MB（我在创建磁盘时给的大小是1GB，由于操作系统之间计算容量的差别，所以存在一定的误差），一共有130个柱面，而且没有分区（提示Disk/dev/sdb doesn't contain a valid partition table），如图4-7所示。

```
[root@localhost ~]# fdisk -l

Disk /dev/sda: 21.4 GB, 21474836480 bytes
255 heads, 63 sectors/track, 2610 cylinders
units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1  *           1           13       104391    83  Linux
/dev/sda2                14        2610     20860402+   8e  Linux LVM

Disk /dev/sdb: 1073 MB, 1073741824 bytes
255 heads, 63 sectors/track, 130 cylinders
units = cylinders of 16065 * 512 = 8225280 bytes

Disk /dev/sdb doesn't contain a valid partition table
[root@localhost ~]#
```

图4-7 查看新的磁盘设备

下面开始对/dev/sdb进行分区操作，首先输入fdisk/dev/sdb，然后输入字母n，这个字母代表new，也就是新建分区；然后系统会提示是创建扩展分区（extended）还是主分区（primary partition），这里选择p；在partition number中输入数字1，代表这是第一个分区；下面要输入第一个柱面开始的位置，该处输入1；然后输入最后一个柱面的位置，这里输入130表示将所有的空间划给这个分区；最后输入字母w，代表将刚刚创建的分区写入分区表。这样就完成了第一步分区操作，所有操作步骤如图4-8所示。

```
[root@localhost ~]#
[root@localhost ~]# fdisk /dev/sdb

Command (m for help): n
Command action
  e   extended
  p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-130, default 1): 1
Last cylinder or +size or +sizeM or +sizeK (1-130, default 130): 130
Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table.
Syncing disks.
[root@localhost ~]#
```

图4-8 分区步骤

分区完成后，再使用fdisk-l查看一下，对比看一下图4-7，

发现不同了吗？是的，这里显示出一个设备，叫做/dev/sdb1，这就是用于下一步创建文件系统的设备。

```
[root@localhost ~]# fdisk -l

Disk /dev/sda: 21.4 GB, 21474836480 bytes
255 heads, 63 sectors/track, 2610 cylinders
units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1    *           1           13        104391   83  Linux
/dev/sda2             14          2610       20860402+   8e  Linux LVM

Disk /dev/sdb: 1073 MB, 1073741824 bytes
255 heads, 63 sectors/track, 130 cylinders
units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1             1           130        1044193+   83  Linux

[root@localhost ~]#
[root@localhost ~]#
[root@localhost ~]#
```

图4-9 确认磁盘分区成功

然后在刚刚创建的分区中格式化文件系统，这里使用的是ext3文件系统。可以使用命令mkfs-t ext3/dev/sdb1，或简单地将此命令写成mkfs.ext3/dev/sdb1，这两个命令是一样的，如图4-10所示。

```
[root@localhost ~]# mkfs.ext3 /dev/sdb1
mke2fs 1.39 (29-May-2006)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
130560 inodes, 261048 blocks
13052 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=268435456
8 block groups
32768 blocks per group, 32768 fragments per group
16320 inodes per group
Superblock backups stored on blocks:
    32768, 98304, 163840, 229376

writing inode tables: done
Creating journal (4096 blocks): done
writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 23 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
[root@localhost ~]#
```

图4-10 创建文件系统

4.2.2 磁盘挂载: mount

创建了文件系统的分区后，在Linux系统下还需要经过挂载才能使用，挂载设备的命令是mount，使用方法如下（其中DEVICE是指具体的设备，MOUNT_POINT是指挂载点，挂载点只能是目录，所以首先在/root目录下创建一个newDisk目录）。

```
[root@localhost ~]# mount DEVICE MOUNT_POINT
[root@localhost ~]# mkdir newDisk
#
挂载设备
[root@localhost ~]# mount /dev/sdb1 newDisk
#
没有参数的mount
会显示所有挂载
[root@localhost ~]# mount
/dev/mapper/VolGroup00-LogVol00 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/sda1 on /boot type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
/dev/sdb1 on /root/newDisk type ext3 (rw) #
挂载成功
#
查看可用空间
[root@localhost newDisk]# df -h | grep sdb1
/dev/sdb1          1004M   18M  936M    2% /root/newDisk
```

4.2.3 设置启动自动挂载：/etc/fstab

前面已经完成了设备的分区、创建文件系统和挂载等操作，是否一切都完成了？别高兴得太早了，因为挂载只是暂时的——之前使用mount命令挂载的设备在你重启计算机之后就会消失，所以必须通过配置/etc/fstab使得系统在重启后能自动挂载。这里只需要如下一条命令即可：

```
echo "/dev/sdb1 /root/newDisk ext3 defaults 0 0" >>/etc/fstab
```

这行命令的意思显而易见：/dev/sdb1（第一部分）挂载到/root/newDisk（第二部分），文件系统是ext3（第三部分），使用系统默认的挂载参数（第四部分defaults），第五部分是决定dump命令在进行备份时是否要将这个分区存档，默认设0，第六部分是设定系统启动时是否对该设备进行fsck，这个值只可能是3种：1保留给根分区，其他分区使用2（检查完根分区后检查）或者0（不检查）。这样以后系统重启时，设备就会自动挂载了。

4.2.4 磁盘检验：fsck、badblocks

当磁盘出现逻辑错误时，可以使用fsck来尝试修复。出现此类错误比较典型的情况是当机器突然掉电时可能引发。该命令的典型使用方式如下（其中TYPE可以是ext2、ext3，最后接设备的全路径）：

```
[root@localhost ~]# fsck -t TYPE /DEVICE/PATH
```

需要特别说明的是，fsck在检查磁盘的时候，需要磁盘是未挂载的状态，否则会造成文件系统损坏。对于已挂载的设备需要先进行umount（解除挂载）操作，umount命令的参数可以是设备路径或者是挂载点，如下所示：

```
[root@localhost ~]# umount /DEVICE/PATH
#
或者
[root@localhost ~]# umount MOUNT_POINT
```

举个例子演示一下如何fsck磁盘，以上一节中挂载的/dev/sdb1为例。如果当前是挂载状态，则需要先进行umount操作，如果umount成功，系统将不会有任何提示，如果umount失败，系统会有相应的报错信息，如下所示：

```
[root@localhost ~]# umount /dev/sdb1
[root@localhost ~]# #
这里没有任何报错，说明umount
成功
```

上面的这个卸载操作的另一种方式如下，效果相同。

```
[root@localhost ~]# umount /root/newDisk
```

卸载完成后，就可以执行fsck了。

```
[root@localhost ~]# fsck -t ext3 /dev/sdb1
fsck 1.39 (29-May-2006)
e2fsck 1.39 (29-May-2006)
/dev/sdb1: clean, 11/130560 files, 8529/261048 blocks
```

想象一下，如果系统的根文件系统出现问题需要fsck怎么办呢？因为系统在运行时，根是无法被umount的。这时候只有重新启动计算机，因为如果确认根文件系统出现问题，系统在重启的时候会检测到这个问题，然后提示用户输入root的密码进入单用户模式，这样就可以使用fsck来修复根文件系统了。

与fsck不同，badblocks主要是用来检测磁盘的物理坏道的，使用这个命令其实更多的只是确认磁盘是否有坏道，所以平时使用得较少，往往只是在怀疑磁盘有坏道的时候才使用。命令如下所示：

```
[root@localhost ~]# badblocks -v /dev/sdb1
Checking blocks 0 to 1044193
Checking          for          bad          blocks          (read-
only test): done
Pass completed, 0 bad blocks found.
```

4.3 Linux逻辑卷

磁盘一旦经过分区后，再想改变磁盘中这个分区的大小就很难了。也就是说，在一个分区经过挂载使用后，随着存储文件的不断增多，可用空间越来越小。如果出现了原先分配的磁盘空间不够使用的情况，这时候是没有办法扩大这个分区的。既然直接使用物理卷的方式无法解决这个问题，那就只能靠分区的时候预估每个分区可能在后期使用中的容量，并划分足够的磁盘空间来最大限度地延迟这个情况的发生了。但是俗话说，计划赶不上变化，也许预估使用的比较多的分区后来实际使用得很少，而预估用得比较少的分区却又需要大量的空间。为了更好地使用磁盘空间，提高系统空间的可扩展性，此时就需要使用逻辑卷。

4.3.1 什么是逻辑卷

逻辑卷就是使用逻辑卷组管理（Logic Volume Manager）创建出来的设备，也是Linux操作系统可以认识的设备。事实上，LVM是介于硬盘裸设备和文件系统的中间层，这种说法比较抽象，不太好理解，要想搞清楚这个问题，首先需要引入逻辑卷组管理中的一些概念，下面来一起看看。

- 物理卷（Physical Volume，PV），也就是物理磁盘分区，比如说/dev/sdb1。如果要想使用LVM来管理这个物理卷，可使用fdisk工具将其ID改为LVM可以识别的值（也就是8e，稍后会做演示）。

- 卷组（Volume Group，VG），也就是PV的集合。

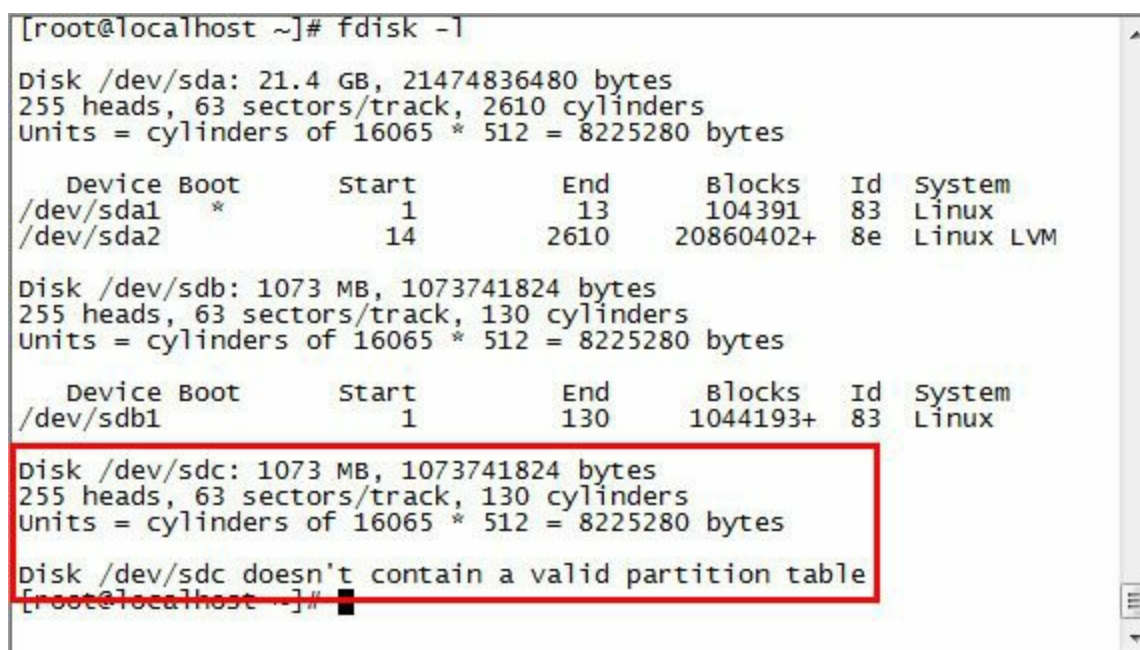
- 逻辑卷（Logic Volume，LV），也就是PV中划出来的一块逻辑磁盘。

了解了概念之后，它们之间的关系就很清晰了：首先创建一个或多个物理卷，物理卷按照相同（或不同）的组名称聚集形成一个（或多个）物理卷组，而逻辑卷就是从某个物理卷组中抽象出来的一块磁盘空间。

4.3.2 如何制作逻辑卷

1. 创建物理卷：pvcreate、pvdisplay

这里继续使用虚拟机来演示逻辑卷的创建过程，首先给虚拟机创建一个大小为1GB的磁盘，操作过程请参考4.2.1中的方法，这里不赘述。添加成功后启动虚拟机，可以发现多了一个/dev/sdc设备，如图4-11所示。



```
[root@localhost ~]# fdisk -l

Disk /dev/sda: 21.4 GB, 21474836480 bytes
255 heads, 63 sectors/track, 2610 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1  *           1           13        104391   83   Linux
/dev/sda2                14         2610       20860402+  8e   Linux LVM

Disk /dev/sdb: 1073 MB, 1073741824 bytes
255 heads, 63 sectors/track, 130 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1                1          130        1044193+   83   Linux

Disk /dev/sdc: 1073 MB, 1073741824 bytes
255 heads, 63 sectors/track, 130 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

Disk /dev/sdc doesn't contain a valid partition table
[root@localhost ~]#
```

图4-11 确认新磁盘设备

将/dev/sdc分成3个区，
即/dev/sdc1（300MB）、/dev/sdc2（300MB）、/dev/sdc3（剩余所有空间）。第一个分区/dev/sdc1的设置步骤如图4-12所示（中间打框的部分演示了如何自定义分区的大小，请读者注意其中的不同）。图4-13和图4-14分别演示了如何创建第二个分区和第三个分区。

```
[root@localhost ~]# fdisk /dev/sdc
Device contains neither a valid DOS partition table, nor Sun, SGI or OS
F disklabel
Building a new DOS disklabel. Changes will remain in memory only,
until you decide to write them. After that, of course, the previous
content won't be recoverable.

Warning: invalid flag 0x0000 of partition table 4 will be corrected by
w(rite)

Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-130, default 1): 1
Last cylinder or +size or +sizeM or +sizeK (1-130, default 130): +300M
Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table.
Syncing disks.
[root@localhost ~]#
```

图4-12 创建第一个分区

```
[root@localhost ~]# fdisk /dev/sdc

Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 2
First cylinder (38-130, default 38): 38
Last cylinder or +size or +sizeM or +sizeK (38-130, default 130): +300M

Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table.
Syncing disks.
[root@localhost ~]#
```

图4-13 创建第二个分区


```
[root@localhost ~]# fdisk /dev/sdc
Command (m for help): n
Command action
  e   extended
  p   primary partition (1-4)
p
Partition number (1-4): 3
First cylinder (75-130, default 75):
Using default value 75
Last cylinder or +size or +sizeM or +sizeK (75-130, default 130):
Using default value 130

Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table.
Syncing disks.
[root@localhost ~]#
```

图4-14 创建第三个分区

分区创建完之后，使用fdisk-l确认一下分区是否确实创建成功了，如图4-15所示。不过当前各个分区的ID值是83，还需要更改ID值为8e，表明该分区是一个特殊的用于逻辑卷管理的分区。具体操作方式如图4-16所示。请读者自行完成第二个、第三个分区的设置。全部修改完后，/dev/sdc的分区信息如图4-17所示。

```
[root@localhost ~]# fdisk -l

Disk /dev/sda: 21.4 GB, 21474836480 bytes
255 heads, 63 sectors/track, 2610 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks    Id  System
/dev/sda1 *           1           13        104391    83  Linux
/dev/sda2             14        2610     20860402+   8e  Linux LVM

Disk /dev/sdb: 1073 MB, 1073741824 bytes
255 heads, 63 sectors/track, 130 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks    Id  System
/dev/sdb1           1         130       1044193+   83  Linux

Disk /dev/sdc: 1073 MB, 1073741824 bytes
255 heads, 63 sectors/track, 130 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks    Id  System
/dev/sdc1           1           37        297171    83  Linux
/dev/sdc2          38           74        297202+   83  Linux
/dev/sdc3          75          130        449820    83  Linux
[root@localhost ~]#
```

图4-15 显示分区

```
[root@localhost ~]# fdisk /dev/sdc
Command (m for help): t 字母t可以修改分区代码
Partition number (1-4): 1 要修改第一个分区
Hex code (type L to list codes): L 如果不记得使用什么代码，可以使用字母L查看

 0 Empty                1e Hidden w95 FAT1      80 old Minix            bf Solaris
 1 FAT12                 24 NEC DOS              81 Minix / old Lin     c1 DRDOS/sec (FAT-
 2 XENIX root            39 Plan 9                82 Linux swap / So    c4 DRDOS/sec (FAT-
 3 XENIX usr              3c PartitionMagic       83 Linux                c6 DRDOS/sec (FAT-
 4 FAT16 <32M            40 Venix 80286           84 OS/2 hidden C:     c7 Syrix
 5 Extended               41 PPC PReP Boot        85 Linux extended     da Non-FS data
 6 FAT16                  42 SFS                   86 NTFS volume set    db CP/M / CTOS / .
 7 HPFS/NTFS              4d QNX4.x                87 NTFS volume set    de Dell utility
 8 AIX                    4e QNX4.x 2nd part     88 Linux plaintext    df BootIt
 9 AIX bootable           4f QNX4.x 3rd part     8e Linux LVM          e1 DOS access
a OS/2 Boot Manag       50 OnTrack DM            93 Amoebe              e3 DOS R/O
b w95 FAT32              51 OnTrack DM6 Aux     94 Amoebe BBT          e4 SpeedStor
c w95 FAT32 (LBA)       52 CP/M                  9f BSD/OS              eb BeOS fs
e w95 FAT16 (LBA)       53 OnTrack DM6 Aux     a0 IBM Thinkpad hi    ee EFI GPT
f w95 Ext'd (LBA)       54 OnTrackDM6           a5 FreeBSD             ef EFI (FAT-12/16/
10 OPUS                  55 EZ-Drive             a6 OpenBSD            f0 Linux/PA-RISC b
11 Hidden FAT12          56 Golden Bow           a7 NextSTEP           f1 SpeedStor
12 Compaq diagnost      5c Priam Edisk          a8 Darwin UFS         f4 SpeedStor
14 Hidden FAT16 <3      61 SpeedStor            a9 NetBSD             f2 DOS secondary
16 Hidden FAT16          63 GNU HURD or Sys     ab Darwin boot        fb VMWare VMFS
17 Hidden HPFS/NTF      64 Novell Netware       b7 BSDI fs            fc VMWare VMKCORE
18 AST SmartSleep       65 Novell Netware       b8 BSDI swap          fd Linux raid auto
1b Hidden w95 FAT3      70 DiskSecure Mult     bb Boot wizard hid    fe LANstep
1c Hidden w95 FAT3      75 PC/IX                be Solaris boot       ff BBT

Hex code (type L to list codes): 8e 输入8e
Changed system type of partition 1 to 8e (Linux LVM)

Command (m for help): w 将修改写入分区表
The partition table has been altered!

calling ioctl() to re-read partition table.
Syncing disks.
[root@localhost ~]#
```

图4-16 修改分区代码

```
Disk /dev/sdc: 1073 MB, 1073741824 bytes
255 heads, 63 sectors/track, 130 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks    Id  System
/dev/sdc1             1           37       297171    8e  Linux LVM
/dev/sdc2            38           74       297202+    8e  Linux LVM
/dev/sdc3            75          130       449820    8e  Linux LVM
[root@localhost ~]#
```

图4-17 修改代码后的分区信息

经过上面的操作，/dev/sdc1、/dev/sdc2、/dev/sdc3就具备了成为PV的条件，下面使用命令pvcreate将分区/dev/sdc1、/dev/sdc2创建为PV，完成后请读者执行pvscan查

看系统所有的物理卷。

```
[root@localhost ~]# pvcreate /dev/sdc1
Physical volume "/dev/sdc1" successfully created
[root@localhost ~]# pvcreate /dev/sdc2
Physical volume "/dev/sdc2" successfully created
```

图4-18 创建PV

虽然使用pvscan命令可以查看系统中的PV，但是显示的内容比较简单，而pvdisplay可以更详细地显示PV的使用状态，因此这里使用的是该命令，如图4-19所示。

```
[root@localhost ~]# pvdisplay
--- Physical volume ---
PV Name               /dev/sda2
VG Name               VolGroup00
PV Size               19.89 GB / not usable 19.49 MB
Allocatable           yes (but full)
PE Size (KByte)       32768
Total PE              636
Free PE               0
Allocated PE          636
PV UUID               kEQiK1-FsOA-mYGB-FETO-20Lt-3Jwh-KlylyR

"/dev/sdc1" is a new physical volume of "290.21 MB"
--- NEW Physical volume ---
PV Name               /dev/sdc1
VG Name
PV Size               290.21 MB
Allocatable           NO
PE Size (KByte)       0
Total PE              0
Free PE               0
Allocated PE          0
PV UUID               Zt1xiJ-xtAX-FmxU-3IX6-H09z-lZZ8-FHAYgw

"/dev/sdc2" is a new physical volume of "290.24 MB"
--- NEW Physical volume ---
PV Name               /dev/sdc2
VG Name
PV Size               290.24 MB
Allocatable           NO
PE Size (KByte)       0
Total PE              0
Free PE               0
Allocated PE          0
PV UUID               wPs2G3-kx39-086A-VYek-xU9H-b8KD-Seq0eb

[root@localhost ~]#
```

图4-19 查看PV

2.创建并查询卷组：vgcreate、vgdisplay

有了PV就可以创建卷组了。命令vgcreate的用法如下（其中VG_NAME是创建的VG名，DEVICE1...DEVICEn代表有多个设备）：

```
[root@localhost ~]# vgcreate VG_NAME DEVICE1 ... DEVICEn
```

现在使用/dev/sdc1、/dev/sdc2创建一个名为First_VG的卷组，如图4-20所示。

可使用vgscan命令来搜索当前系统上的所有VG，还可使用vgdisplay可以看到更详细的信息，如图4-21所示。

```
[root@localhost ~]# vgcreate First_VG /dev/sdc1 /dev/sdc2
Volume group "First_VG" successfully created
```

图4-20 创建VG

```
[root@localhost ~]# vgscan
Reading all physical volumes. This may take a while...
Found volume group "First_VG" using metadata type lvm2
Found volume group "VolGroup00" using metadata type lvm2
[root@localhost ~]# vgdisplay
--- Volume group ---
VG Name                First_VG      这是刚刚创建的First_VG
System ID
Format                 lvm2
Metadata Areas         2
Metadata Sequence No   1
VG Access              read/write
VG Status              resizable
MAX LV                 0
Cur LV                0
Open LV               0
Max PV                 0
Cur PV                2
Act PV                 2
VG Size                576.00 MB
PE Size                4.00 MB
Total PE               144
Alloc PE / Size        0 / 0
Free PE / Size         144 / 576.00 MB
VG UUID                iqGmsL-4iR1-VQab-mIGU-GjPG-3902-2sPY9Q
```

图4-21 查找VG

在图4-21中，First_VG大小为576MB，等于/dev/sdc1、/dev/sdc2这两个PV大小之和（分区和制作PV、

VG的过程中会消耗一部分磁盘空间）。

3.扩容卷组：vgextend

如果在使用过程中发现要扩大First_VG，可以使用vgextend随时扩大VG的容量，其中VG_NAME是需要增加的VG名，DEVICE1...DEVICEn代表是多个设备。

```
[root@localhost ~]# vgextend VG_NAME DEVICE1 ... DEVICEn
```

现在把之前创建的/dev/sdc3的分区加入First_VG中：首先需要将/dev/sdc3做成PV，然后再使用vgextend扩大VG的容量，如图4-22所示。



```
[root@localhost ~]#  
[root@localhost ~]# pvcreate /dev/sdc3 先将/dev/sdc3创建成PV  
Physical volume "/dev/sdc3" successfully created  
[root@localhost ~]# vgextend First_VG /dev/sdc3 将/dev/sdc3加入First_VG  
Volume group "First_VG" successfully extended  
[root@localhost ~]# vgdisplay  
--- volume group ---  
VG Name                First_VG  
System ID  
Format                 lvm2  
Metadata Areas         3  
Metadata Sequence No   2  
VG Access               read/write  
VG Status               resizable  
MAX LV                  0  
Cur LV                 0  
Open LV                 0  
Max PV                  0  
Cur PV                 3  
Act PV                  3  
VG Size                 1012.00 MB 容量增大为1G了，成功扩容First_VG  
PE Size                 4.00 MB  
Total PE                253  
Alloc PE / Size         0 / 0  
Free PE / Size          253 / 1012.00 MB  
VG UUID                 iqGmsL-4iR1-VQab-mIgu-GjPG-3902-2sPY9Q
```

图4-22 VG扩容

4.创建逻辑卷：lvcreate、lvdisplay

有了卷组（First_VG），就可以创建逻辑卷了。在讲具体

的命令之前，自然而然会想到，在创建逻辑卷的时候需要定义逻辑卷的大小、名称，以及该逻辑卷使用的是哪个卷组的空间等信息，`lvcreate`命令可以完成这些工作。其中，`-L`指定逻辑卷的大小，后跟的`SIZE`表示具体的逻辑卷大小的值，比如说100MB，`-n`为指定逻辑卷的名字，最后的参数`VG_NAME`是指定从什么卷组中分配空间，如下所示：

```
[root@localhost ~]# lvcreate -L SIZE -n LV_NAME VG_NAME
```

按照命令的使用方式，创建一个大小为100MB的逻辑卷，命名为`First_LV`，所用空间从`First_VG`中划分。创建完成后使用`lvdisplay`查看一下`First_LV`的使用情况，如图4-23所示。

```
[root@localhost ~]# lvcreate -L 100M -n First_LV First_VG
Logical volume "First_LV" created
[root@localhost ~]# lvdisplay
--- Logical volume ---
LV Name                /dev/First_VG/First_LV
VG Name                First_VG
LV UUID                4So008-05zk-wBuI-9YAi-MByS-Dexq-bVdfYi
LV write Access        read/write
LV Status              available
# open                 0
LV Size                100.00 MB
Current LE             25
Segments              1
Allocation             inherit
Read ahead sectors     auto
- currently set to    256
Block device           253:2
```

图4-23 创建LV

5.创建文件系统并挂载

现在我们已经成功创建了一个逻辑卷，但是目前还无法使用它。和使用物理分区一样，逻辑卷也需要在创建文件系统、挂载后才能被系统使用，需要说明的是，在对逻辑卷创建文件系统的时候，其全路径是`/dev/卷组名/逻辑卷名`，具体操作步骤方式如图4-24所示。

```
[root@localhost ~]# mkfs.ext3 /dev/First_VG/First_LV 创建文件系统
mke2fs 1.39 (29-May-2006)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
25688 inodes, 102400 blocks
5120 blocks (5.00%) reserved for the super user
First data block=1
Maximum filesystem blocks=67371008
13 block groups
8192 blocks per group, 8192 fragments per group
1976 inodes per group
Superblock backups stored on blocks:
    8193, 24577, 40961, 57345, 73729

writing inode tables: done
Creating journal (4096 blocks): done
writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 25 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
[root@localhost ~]# mkdir /root/newLV 创建一个挂载点
[root@localhost ~]# mount /dev/First_VG/First_LV /root/newLV 挂载
[root@localhost ~]# █
```

图4-24 对LV创建文件系统

4.4 硬链接和软链接

4.4.1 什么是硬链接

硬链接（hard link）又称实际链接，是指通过索引节点来进行链接。在Linux文件系统中，所有的文件都会有一个编号，称为inode，多个文件名指向同一索引节点是被允许的，这种链接就是硬链接。硬链接的作用是允许一个文件拥有多个有效路径名，这样用户就可以建立硬链接指向同一文件，删除一个链接并不会影响索引节点本身和其他的链接，只有当最后一个链接被删除时，文件的数据块及目录的链接才会被释放。也就是说，文件真正删除的前提条件是与之相关的所有硬链接均被删除。硬链接有两个限制：

- 不允许给目录创建硬链接；
- 只有在同一文件系统中的文件之间才能创建链接，即不同分区上的两个文件之间不能够建立硬链接。

下面让我们看一下如何创建一个硬链接。

```
#
进入/root
目录
[root@localhost ~]# cd
#
创建hard
目录
[root@localhost ~]# mkdir hard
#
进入hard
目录
[root@localhost ~]# cd hard
#
创建一个文件
```

```
[root@localhost hard]# touch hard01
#ls
后的-i
参数可以显示文件的inode
, 此处显示3834061
[root@localhost hard]# ls -li
total 0
3834061 -rw-r--r-- 1 root root 0 Jan 15 10:50 hard01
#
创建指向hard01
的硬链接hard01_hlink
[root@localhost hard]# ln hard01 hard01_hlink
#
硬链接hard01_hlink
指向的inode
和hard01
指向的inode
值是一致的
[root@localhost hard]# ls -li
total 0
3834061 -rw-r--r-- 2 root root 0 Jan 15 10:50 hard01
3834061 -rw-r--r-- 2 root root 0 Jan 15 10:50 hard01_hlink
```

以上演示了硬链接的方法。注意，在创建硬链接的前后分别使用ls-li命令，你能发现hard01的输出有什么不同吗？答案是第三列的值变化了！这个值其实是源文件的关联数，文件创建之初该值为1，该文件每增加一个硬链接该值将增1，当此数为0的时候该文件才能真正被文件系统删除。

4.4.2 什么是软链接

软链接（soft link）又称符号链接（symbolic link），是一个包含了另一个文件路径名的文件，可以指向任意文件或目录，也可以跨不同的文件系统。软链接和Windows下的“快捷方式”十分类似，删除软链接并不会删除其所指向的源文件，如果删除了源文件则软链接会出现“断链”。

```
#
进入/root
目录
[root@localhost ~]# cd
[root@localhost ~]# mkdir soft
[root@localhost ~]# cd soft
[root@localhost soft]# touch file01
[root@localhost soft]# ln -s file01 file01_slink
#
创建软链接，使用了-s
参数
[root@localhost soft]# ls -li
total 0
3834063 -rw-r--r-- 1 root root 0 Jan 15 11:14 file01
[root@localhost soft]# ln -s file01 file01_slink
[root@localhost soft]# ls -li
total 0
3834063 -rw-r--r-- 1 root root 0 Jan 15 11:14 file01
3834064 lrwxrwxrwx 1 root root 6 Jan 15 11:14 file01_slink -
> file01
```

创建软链接需要使用-s参数。另外还请注意，在创建软链接的前后分别使用ls-li命令，会发现软链接的inode和源文件的inode不一样，这说明软链接本身就是一个文件。

读者可以尝试删除软链接的源文件，然后可以在终端中看到对应的软链接将会以闪烁的方式标记其已是一个断链。

第5章 字符处理

5.1 管道

说起“管道”，很容易让人想起现实生活中使用的水管、输气管等，它们的作用在于运输气体或液体等物质，有了管道，会让我们方便很多。在Linux中也存在着管道，它是一个固定大小的缓冲区，该缓冲区的大小为1页，即4K字节。管道是一种使用非常频繁的通信机制，我们可以用管道符“|”来连接进程，由管道连接起来的进程可以自动运行，如同有一个数据流一样，所以管道表现为输入输出重定向的一种方法，它可以把一个命令的输出内容当作下一个命令的输入内容，两个命令之间只需要使用管道符连接即可。

举个例子，如果想要看一下/etc/init.d目录下文件的详细信息，可以使用ls-l/etc/init.d命令，不过这可能会出现因输出内容过多而造成刷屏的情况，这样一来，先输出的内容在屏幕上就看不到了。其实这里就可以利用管道功能，将命令的输出使用more程序一页一页地显示出来。

```
[root@localhost ~]# ls -l /etc/init.d | more
```

可以看出，通过管道，使ls-l/etc/init.d命令输出的内容作为下一个命令more的输入，这样就可以方便地查看输出内容了。

5.2 使用grep搜索文本

grep是Linux下非常强大的基于行的文本搜索工具，使用该工具时，如果匹配到相关信息就会打印出符合条件的所有行。下面列出了该命令常用的参数：

```
[root@localhost ~]# grep [-ivnc] '
需要匹配的字符'
文件名
#-i
不区分大小写
#-c
统计包含匹配的行数
#-n
输出行号
#-v
反向匹配
```

为演示grep的用法，这里首先创建一个文件，文件名和文件内容如下：

```
[root@localhost ~]# cat tomAndJerry.txt
The cat's name is Tom, what's the mouse's name?
The mouse's NAME is Jerry
They are good friends
```

下面要找出含有name的行：

```
[root@localhost ~]# grep 'name' tomAndJerry.txt
The cat's name is Tom, what's the mouse's name?
#
打印出含有name
的行的行编号
[root@localhost ~]# grep -n 'name' tomAndJerry.txt
1:The cat's name is Tom, what's the mouse's name?
```

由于grep区分大小写，所以虽然第二行中含有大写的NAME，但是也不会匹配到。如果希望忽略大小写，可以加上-i参数。

```
[root@localhost ~]# grep -i 'name' tomAndJerry.txt
The cat's name is Tom, what's the mouse's name?
The mouse's Name is Jerry
```

如果想知道文件中一共有多少包含name的行，可以使用下面的命令。注意到第二条命令和第一条命令只有一个参数的差别，但是输出的结果却是不一样的。了解了-i参数的作用就不难理解了，请读者自行区分以下两条命令的区别。

```
[root@localhost ~]# grep -c 'name' tomAndJerry.txt
1
[root@localhost ~]# grep -ci 'name' tomAndJerry.txt
2
```

如果想打印出文件中不包含name的行，可以使用grep的反选参数-v。读者可自行区分以下命令的区别：

```
[root@localhost ~]# grep -v 'name' tomAndJerry.txt
The mouse's Name is Jerry
They are good friends
[root@localhost ~]# grep -vi 'name' tomAndJerry.txt
They are good friends
```

以上命令都可以使用cat命令+管道符改写。比如上一个命令可以这样改写：

```
[root@localhost ~]# cat tomAndJerry.txt | grep -vi 'name'
```

They are good friends

5.3 使用sort排序

很多情况下需要对无序的数据进行排序，这时就要用到sort排序了。下面列出了该命令常用的参数：

```
[root@localhost ~]# sort [-ntkr]  
文件名  
#-n  
采取数字排序  
#-t  
指定分隔符  
#-k  
指定第几列  
#-r  
反向排序
```

为演示sort的用法，这里首先创建一个文件，文件名和文件内容如下：

```
[root@localhost ~]# cat sort.txt  
b:3  
c:2  
a:4  
e:5  
d:1  
f:11
```

下面对输出的内容进行排序：

```
[root@localhost ~]# cat sort.txt | sort  
a:4  
b:3  
c:2  
d:1  
e:5
```

```
f:11
```

```
#
```

对输出内容直接排序时，默认按照每行的第一个字符进行排序

下面表示对输出内容进行反向排序：

```
[root@localhost ~]# cat sort.txt | sort -r
```

```
f:11
```

```
e:5
```

```
d:1
```

```
c:2
```

```
b:3
```

```
a:4
```

```
#
```

和上面的例子的输出相反

可观察到，`sort.txt`文件具有一个特点，第一个字符是字母，第三个字符是数字，中间用冒号隔开。这样就可以用`-t`指定分隔符，并用`-k`指定用于排序的列了。

```
[root@localhost ~]# cat sort.txt | sort -t ":" -k 2
```

```
d:1
```

```
f:11
```

```
c:2
```

```
b:3
```

```
a:4
```

```
e:5
```

```
#
```

你可能已注意到，当前的排序是按照数字列的部分进行的。不过，第2行为什么是11

呢？

那是因为当前的排序并不是按照“数字值来进行的”

在上面的命令中，当前的排序是按照以冒号隔开的第二部分进行的，不过读者是否注意到，第二行是`f:11`，这一行不应该在最后一行吗？因为11是最大的。但其实命令的输出并不是错误的，因为按照排序的方式，只会看第一个字符，而11第一

个字符是1，按照字符来排序那它确实比2小。如果想要指定按照“数字”的方式进行排序，则需要加上-n参数。

```
[root@localhost ~]# cat sort.txt | sort -t ":" -k 2 -n  
d:1  
c:2  
b:3  
a:4  
e:5  
f:11
```

5.4 使用uniq删除重复内容

如果文件（或标准输出）中有多行完全相同的内容，我们很自然希望能删除重复的行，同时还可以统计出完全相同的行出现的总次数，`uniq`命令就能帮助解决这个问题。下面列出了该命令常用的参数：

```
[root@localhost ~]# uniq [-ic]
#-i
忽略大小写
#-c
计算重复行数
```

为演示`uniq`的用法，这里首先创建一个文件，文件名和文件内容如下：

```
[root@localhost ~]# cat uniq.txt
abc
123
abc
123
```

需要说明的是，`uniq`一般都需要和`sort`命令一起使用，也就是先将文件使用`sort`进行排序（这样重复的内容就能显示在连续的几行中），然后再使用`uniq`删除掉重复的内容（`uniq`的作用就在于删除连续的完全一致的行）。读者观察一下以下两次命令的输出，第一次直接`cat`输出文件，然后使用`uniq`命令，输出的内容居然和原文件`uniq.txt`的内容是一样的，这是因为`uniq`命令只会对比相邻的行，如果有连续相同的若干行则删除重复内容，仅输出一行。如果相同的行非连续，则`uniq`命令不具备删除效果。第二次则在使用`sort`排序后再使用`uniq`命令，这时就达到了预期的效果。

```
[root@localhost ~]# cat uniq.txt | uniq
abc
123
abc
123
[root@localhost ~]# cat uniq.txt | sort | uniq
123
abc
#
使用 -c
参数就会在每行前面打印出该行重复的次数
[root@localhost ~]# cat uniq.txt | sort | uniq -c
2 123
2 abc
```

5.5 使用cut截取文本

顾名思义，`cut`就是截取的意思，它能处理的对象是“一行”文本，可从中选取出用户所需要的部分。在有特定的分隔符时，可以指定分隔符，然后打印出以分隔符隔开的具体某一行或某几列，这里`cut`的用法如下：

```
[root@localhost ~]# cut -f  
指定的列 -d'  
分隔符'
```

举个例子，在文件`/etc/passwd`中，每行都是使用6个冒号隔开的7列文本，那么很容易使用`cut`的这个功能来提取出特定的信息。比如说我们需要打印出系统中的所有用户：

```
[root@localhost ~]# cat /etc/passwd | cut -f1 -d':'
```

或者想同时打印出用户和这个用户的家目录：

```
[root@localhost ~]# cat /etc/passwd | cut -f1,6 -d':'  
root:/root  
bin:/bin  
daemon:/sbin  
adm:/var/adm  
.....(  
略去内容).....
```

如果还想同时打印出每位用户的登录shell：

```
[root@localhost ~]# cat /etc/passwd | cut -f1,6-7 -d':'
```

```
root:/root:/bin/bash
bin:/bin:/sbin/nologin
daemon:/sbin:/sbin/nologin
adm:/var/adm:/sbin/nologin
.....(
略去内容).....
```

以上cut使用的场景是在处理的行中有特定分隔符的时候，但如果要处理的行是没有分隔符的，那是不是cut就没有用武之地了？答案是否定的，cut还可以打印指定的字符，这时候cut的用法如下：

```
[root@localhost ~]# cut -c
指定列的字符
```

继续使用/etc/passwd为例子，假设想要打印出每行第1~5个字符，以及第7~10个字符的内容，如下所示：

```
[root@localhost ~]# cat /etc/passwd | cut -c1-5,7-10
root::0:0
bin:x1:1:
daemo:x:2
adm:x3:4:
.....(
略去内容).....
```

5.6 使用tr做文本转换

tr命令比较简单，其主要作用在于文本转换或删除。这里假设要把文件/etc/passwd中的小写字母转换为大写字母，然后再尝试删除文本中的冒号，如下所示：

```
[root@localhost ~]# cat /etc/passwd | tr '[a-z]' '[A-Z]'
ROOT:X:0:0:ROOT:/ROOT:/BIN/BASH
BIN:X:1:1:BIN:/BIN:/SBIN/NOLOGIN
DAEMON:X:2:2:DAEMON:/SBIN:/SBIN/NOLOGIN
ADM:X:3:4:ADM:/VAR/ADM:/SBIN/NOLOGIN
.....(
略去内容).....
[root@localhost ~]# cat /etc/passwd | tr -d ':'
rootx00root/root/bin/bash
binx11bin/bin/sbin/nologin
daemonx22daemon/sbin/sbin/nologin
admx34adm/var/adm/sbin/nologin
.....(
略去内容).....
```

5.7 使用paste做文本合并

paste的作用在于将文件按照行进行合并，中间使用tab隔开。假设有两个文件分别为a.txt、b.txt，下面使用paste命令来合并文件，如下所示：

```
#
文件a.txt
中的内容
[root@localhost ~]# cat a.txt
1
2
3
#
文件b.txt
中的内容
[root@localhost ~]# cat b.txt
a
b
c
#
使用paste
连接这两个文件，可以看到只是将文件按照行做了合并
[root@localhost ~]# paste a.txt b.txt
1      a
2      b
3      c
#
也可以使用 -d
指定在合并文件时行间的分隔符
[root@localhost ~]# paste -d: a.txt b.txt
1:a
2:b
3:c
```

5.8 使用split分割大文件

早几年前，“文件分割”这4个字还是比较流行的，当时受限于移动存储设备的限制，大文件的转移往往需要通过分割成小文件分别存储来实现，之后会再使用合并的方式还原成原始文件。虽然随着现代移动存储、网络存储的发展，分割大文件的做法已经不那么流行了，但是了解一下还是必要的。在Linux下使用split命令来实现文件的分割，支持按照行数分割和按照大小分割这两种模式。要说明的是，二进制文件因为没有“行”的概念，所以二进制文件无法使用行分割，而只能按照文件大小进行分割。相关命令如下所示：

```
#
假设文件中有一个512MB
的大文件
[root@localhost ~]# ll -h big_file.txt
-rw-r--r-- 1 root root 512M Jan 24 14:16 big_file.txt
#
按照行进行分割，-l
参数指定每500
行为一个小文件
[root@localhost ~]# split -l 500 big_file.txt small_file_
#
分割完成后，当前目录下会生成很多小文件
[root@localhost ~]# ls small_file_*
small_file_aa small_file_ab small_file_ac small_file_ad
.....(
略去内容).....
#
如果文件是二进制的，则只能按照文件大小分割
[root@localhost ~]# ll -h big_bin
-rw-r--r-- 1 root root 512M Jan 24 14:51 big_bin
[root@localhost ~]# split -b 64m big_bin small_bin_
#
分割完成后，当前目录下会生成很多大小为64MB
的文件
[root@localhost ~]# ll -h small_bin_*
-rw-r--r-- 1 root root 64M Jan 24 14:53 small_bin_aa
-rw-r--r-- 1 root root 64M Jan 24 14:53 small_bin_ab
```

```
-rw-r--r-- 1 root root 64M Jan 24 14:53 small_bin_ac
-rw-r--r-- 1 root root 64M Jan 24 14:53 small_bin_ad
-rw-r--r-- 1 root root 64M Jan 24 14:53 small_bin_ae
-rw-r--r-- 1 root root 64M Jan 24 14:53 small_bin_af
-rw-r--r-- 1 root root 64M Jan 24 14:53 small_bin_ag
-rw-r--r-- 1 root root 64M Jan 24 14:53 small_bin_ah
.....(
略去内容).....
```

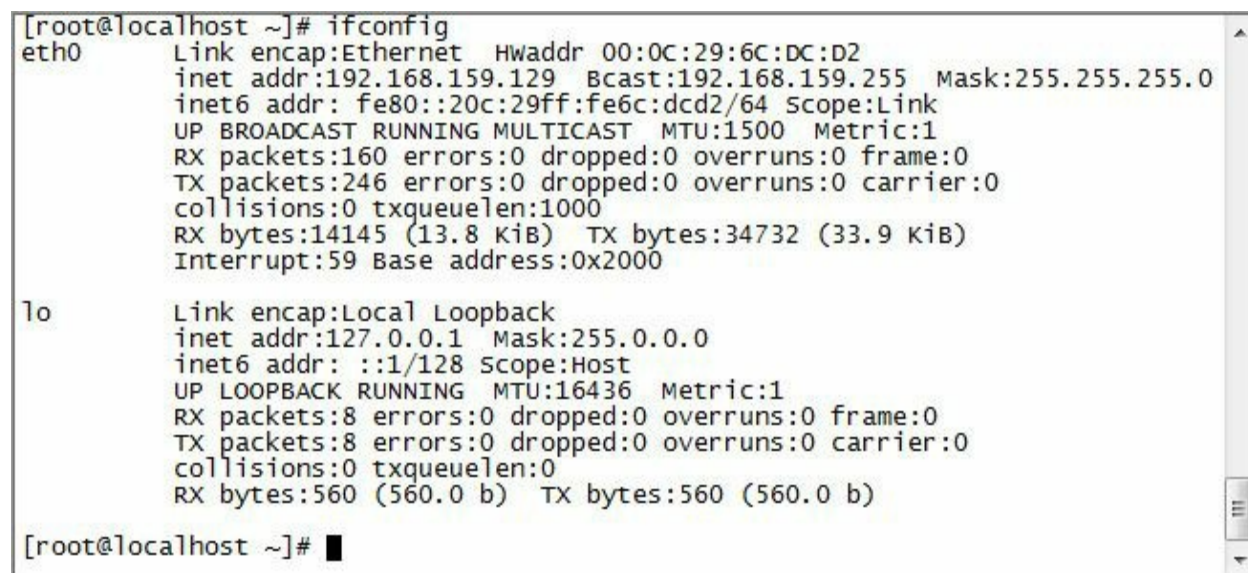
第6章 网络管理

Linux作为一个越来越成熟的系统，在服务器市场、嵌入式设备等方面都取得了巨大的成功，在网络上的应用也越来越多。事实上，从Linux诞生时起，其就被赋予了强大的网络功能，所以掌握如何在Linux系统中配置、管理网络就变得非常必要。第1章在讲述如何安装Linux的过程中，网卡配置部分我们选择的是“从DHCP获得地址”，这样配置后，如果当前网络中存在DHCP服务器，就会自动获得配置参数。不过，如果后期要检查或自主配置网络相关参数，就必须熟练掌握Linux下的相关网络配置命名和方法了。本章就将针对这一部分内容来进行讲解。

6.1 网络接口配置

6.1.1 使用ifconfig检查和配置网卡

如果不使用任何参数，输入ifconfig命令时将会输出当前系统中所有处于活动状态的网络接口，如图6-1所示。



```
[root@localhost ~]# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0C:29:6C:DC:D2
          inet addr:192.168.159.129  Bcast:192.168.159.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe6c:dcd2/64  Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:160 errors:0 dropped:0 overruns:0 frame:0
          TX packets:246 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:14145 (13.8 KiB)  TX bytes:34732 (33.9 KiB)
          Interrupt:59 Base address:0x2000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128  Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:560 (560.0 b)  TX bytes:560 (560.0 b)

[root@localhost ~]#
```

图6-1 不带参数的ifconfig

图6-1中的eth0表示的是以太网的第一块网卡。其中eth是Ethernet的前三个字母，代表以太网，0代表是第一块网卡，第二块以太网网卡则是eth1，以此类推。Link encap是指封装方式为以太网；HWaddr是指网卡的硬件地址（MAC地址）；inet addr是指该网卡当前的IP地址；Broadcast是广播地址（这部分是由系统根据IP和掩码算出来的，一般不需要手工设置）；Mask是指掩码；UP说明了该网卡目前处于活动状态；MTU代表最大存储单元，即此网卡一次所能传输的最大分包；RX和TX分别代表接收和发送的包；collision代表发生的冲突数，如果发现值不为0则很可能网络存在故障；txqueuelen代表传输缓冲区长度大小；第二个设备是lo，表示主机的环回

地址，这个地址是用于本地通信的。

如果在ifconfig命令后面跟上具体设备的名称（比如eth0），则只显示指定设备的相关信息，如图6-2所示。

```
[root@localhost ~]# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:0C:29:6C:DC:D2
          inet addr:192.168.159.129  Bcast:192.168.159.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe6c:dcd2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2521 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1967 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:2566939 (2.4 MiB)  TX bytes:140743 (137.4 KiB)
          Interrupt:59 Base address:0x2000

[root@localhost ~]# █
```

图6-2 带参数的ifconfig

由于某种原因如果希望手工指定eth0的IP地址，那么可按如下方式进行修改：

```
[root@localhost ~]# ifconfig eth0 192.168.159.130 netmask 255
#
```

上面的命令可以简写为：

```
#[root@localhost ~]# ifconfig eth0 192.168.159.130/24
#
```

通过IP地址和掩码系统能自行算出广播地址，也可以显式地指定广播地址，不过一般情况下没有必要这么做

```
[root@localhost ~]# ifconfig eth0 192.168.159.130 broadcast 1
netmask 255.255.255.0
```

有时候需要手工断开/启用网卡，以eth0为例，使用方法如下：

```
[root@localhost ~]# ifconfig eth0 down
#
```

在关闭了网卡后，再使用不加参数的ifconfig命令时，将不再显示eth0

```
#
```

但是可以使用ifconfig

-a

显示所有包括当前不活动的网卡

```
[root@localhost ~]# ifconfig eth0 up
```

#

启动网卡eth0

#

以上关闭和启动网卡的命令等同于如下两条命令

```
#[root@localhost ~]# ifdown eth0
```

```
#[root@localhost ~]# ifup eth0
```

6.1.2 将IP配置信息写入配置文件

上一小节讲到的ifconfig命令可以直接配置网卡IP，但是这属于一种动态的配置，所配置的信息只是保存在当前运行的内核中。一旦系统重启，这些信息将丢失。为了能在重启后依然生效，可以在相关的配置文件中保存这些信息，这样，系统重启后将从这些配置文件中读取出来。RedHat和CentOS系统的网络配置文件所处的目录为/etc/sysconfig/network-scripts/，eth0的配置文件为ifcfg-eth0，如果有第二块物理网卡，则配置文件为ifcfg-eth1，以此类推。

一直以来，书中用于演示的虚拟机的配置文件是安装系统时生成的，内容如下：

```
[root@localhost network-scripts]# cat ifcfg-eth0
# Advanced Micro Devices [AMD] 79c970 [PCnet32 LANCE]
DEVICE=eth0
BOOTPROTO=dhcp
ONBOOT=yes
```

其中，DEVICE变量定义了设备的名称；BOOTPROTO变量定义了获取IP的方式，这里BOOTPROTO=dhcp的含义是：系统在启用这块网卡时，IP将会通过dhcp的方式获得；还有个可选的值是static，表示静态设置的IP；ONBOOT变量定义了启动时是否激活使用该设备，yes表示激活，no表示不激活。

为了静态化地为该系统配置一个IP（这里假设IP为192.168.159.129，子网掩码为255.255.255.0），将配置文件修改如下：

```
[root@localhost network-scripts]# cat ifcfg-eth0
# Advanced Micro Devices [AMD] 79c970 [PCnet32 LANCE]
```

```
DEVICE=eth0
BOOTPROTO=static
ONBOOT=yes
IPADDR=192.168.159.129
NETMASK=255.255.255.0
```

修改完成后，如果要想立即生效，可以将端口先停用再启用，或者重启网络服务。虽然这两种方式的效果是一样的，但是在实际工作中也要注意，第一种方式是不能远程操作的，因为一旦网卡被关闭掉后远程连接就断开了，随后的启用命令也就无法输入了，所以这种方式只能在管理员可以物理地接触到服务器的时候使用（比如说你正坐在被操作的这台服务器面前，使用本地终端操作服务器而不是远程登录）。第二种方式虽然也经过了一次网络断开，但是该命令会在断开后立即启用网络，因此只需要使用新的IP重新连接就可以了。

```
[root@localhost ~]# ifconfig eth0 down
[root@localhost ~]# ifconfig eth0 up
#
或者重启网络服务，也可以立即生效，推荐使用这种方式
#[root@localhost ~]# service network restart
```

6.2 路由和网关设置

Linux主机之间是使用IP进行通信的，假设A主机和B主机同在一个网段内且网卡都处于激活状态，则A具备和B直接通信的能力（通过交换机或简易HUB）。但是如果A主机和B主机处于两个不同的网段，则A必须通过路由器才能和B通信。一般来说，路由器属于IT设备的基础设施，每一个网段都应该有至少一个网关。在Linux中可使用route命令添加默认网关。假设添加的网关是192.168.159.2，添加方式如下：

```
[root@localhost ~]# route add default gw 192.168.159.2
```

在以上命令中，只需要将add改成del，就能删除刚才添加的路由。

```
[root@localhost ~]# route del default gw 192.168.159.2
#
该命令可以简写成如下形式
[root@localhost ~]# route del default
```

添加网关后，可以使用route-n查看系统当前的路由表。

```
[root@localhost ~]# route -n
Kernel IP routing table
Destination      Gateway          Genmask         Flags Metric
192.168.159.0    0.0.0.0         255.255.255.0   U        0
169.254.0.0      0.0.0.0         255.255.0.0     U        0
0.0.0.0          192.168.159.2   0.0.0.0         UG       0
```

同样的，如果只使用route命令添加网关，一旦系统重启，配置信息就不存在了，必须将这种配置信息写到相关的配置文

件中才能永久保存。可以在网卡配置文件中使用GATEWAY变量来定义网关，只需要添加如下部分到ifcfg-eth0中即可，当然别忘了重启网络服务使配置生效。

```
GATEWAY=192.168.159.2
```

另外，在配置文件/etc/sysconfig/network中添加这段配置也能达到同样的效果。

6.3 DNS客户端配置

6.3.1 /etc/hosts

因特网发明初期，联网的主机数量有限，想要访问对方主机时只需要输入对方的IP地址即可。但是随着主机数量的不断增长，单凭人脑已经无法记忆越来越多的IP地址了。为了解决这个问题，人们使用hosts文件来记录主机名和IP的对应关系，这样访问对方的主机时，就不需要使用IP了，只需要使用主机名。这个文件在Linux下就是/etc/hosts，这种方式确实“可以工作”，但是当主机数量增长到一定数量级的时候仍然无法适用。为了彻底解决这个问题，人们发明了DNS系统。经过几十年的发展，虽然系统、网络技术都发生了翻天覆地的变化，但是这个文件还是被当作传统保留了下来。具体来说，hosts文件的作用主要如下：

- 加快域名解析。当访问网站时，系统会首先查看hosts文件中是否有记录，如果记录存在则直接解析出对应的IP，这时则不需要请求DNS服务器。

- 方便小型局域网用户使用的内部设备。很多单位的局域网中都存在着不少内部应用系统（比如办公自动化OA、公司论坛等），平时在工作中也都需要访问，但是由于这些局域网太小而不必为此专门设置DNS服务器，那么此时使用hosts文件则能简单地解决这个问题。

假设公司里有A、B两台主机，B主机的IP为10.1.1.145，为了方便访问B主机，可以在A主机的/etc/hosts文件中添加一条记录：

10.1.1.145	hostB
------------	-------

完成后在A主机上使用ping命令测试到B主机的连通性，在ping的输出中可以看到主机名hostB被正确地解析为10.1.1.145，如果没有之前添加的记录，这里将会显示ping:unknown host hostB的错误。

```
[root@localhost ~]# ping hostB -c 1
PING hostB (10.1.1.145) 56(84) bytes of data.
64 bytes from hostB (10.1.1.145): icmp_seq=1 ttl=64 time=0.79
--- hostB ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.797/0.797/0.797/0.000 ms
```

6.3.2 /etc/resolv.conf

使用hosts文件毕竟只能做有限的主机记录，无法将所有已知的主机名记录到hosts文件中。因此，当今几乎所有的主机都在使用DNS来解析地址，从技术上来说，DNS就是全互联网上主机名及其IP地址对应关系的数据库。设置主机为DNS客户端的配置文件就是/etc/resolv.conf，其中包含nameserver、search、domain这3个关键字。以下是当前笔者测试机上的/etc/resolv.conf文件：

```
[root@localhost ~]# cat /etc/resolv.conf
; generated by /sbin/dhclient-script
search localdomain
nameserver 192.168.159.2
```

nameserver关键字后面紧跟着一个DNS主机的IP地址，可以设置2~3个nameserver，但是主机在查询域名时会首先查询第一个DNS，当该DNS不可用时才会查询第二个DNS，以此类推。注意，虽然你可以在该文件中定义多于3个的nameserver，但是这并没有意义，因为系统永远不会用到第四个nameserver（笔者在CentOS5.5和RedHat5.5中做过测试）。

search关键字后紧跟的是一个域名。每个主机严格来说都应该有一个FQDN（全限定域名），所以往往域名就很长，如果这里写成search google.com，那么www就代表www.google.com了，这个关键字后可以跟多个域名。

domain关键字和search类似，不同的是domain后面只能跟一个域名。

6.4 网络测试工具

6.4.1 ping

ping程序的目的在于测试另一台主机是否可达，一般来说，如果ping不到某台主机，就说明对方主机已经出现了问题，但是不排除由于链路中防火墙的因素、ping包被丢弃等原因而造成ping不通的情况。ping命令最简单的使用方式是接收一个主机名或IP作为其单一的参数，在按回车键后，执行ping命令的主机会向对端主机发送一个ICMP的echo请求包，对端主机在接收到这个包后会回应一个ICMP的reply回应包。在Linux下ping命令并不会主动停止，需要使用Ctrl+C组合键来停止，ping命令将会对发出的请求包和收到的回应包进行计数，这样就能计算网络丢包率。

```
[root@localhost ~]# ping 10.1.1.145
PING 10.1.1.145 (10.1.1.145) 56(84) bytes of data.
64 bytes from 10.1.1.145: icmp_seq=1 ttl=64 time=3.60 ms
64 bytes from 10.1.1.145: icmp_seq=2 ttl=64 time=1.32 ms
64 bytes from 10.1.1.145: icmp_seq=3 ttl=64 time=0.619 ms
64 bytes from 10.1.1.145: icmp_seq=4 ttl=64 time=0.655 ms
[Ctrl+C] #
此处手工输入Ctrl+C
组合键
--- 10.1.1.145 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3072m
rtt min/avg/max/mdev = 0.619/1.551/3.604/1.218 ms
```

表6-1列出了ping命令其他的一些参数。

表6-1 ping命令的常用参数

参数	含 义
-c	指定 ping 的次数
-i	指定 ping 包的发送间隔
-w	如果 ping 没有回应，则在指定超时时间后退出

6.4.2 host

host命令是用来查询DNS记录的，如果使用域名作为host的参数，命令返回该域名的IP，如下所示：

```
[root@localhost ~]# host www.google.com
www.google.com has address 74.125.128.147
www.google.com has address 74.125.128.103
www.google.com has address 74.125.128.99
www.google.com has address 74.125.128.104
www.google.com has address 74.125.128.105
www.google.com has address 74.125.128.106
www.google.com has IPv6 address 2404:6800:4005:c00::67
```

大家试一下在浏览器中直接输入任意一个查询到的IP地址，按回车键后是不是可以看到google的主页了？以上命令还可以有第二个参数，该参数必须是一个可用的DNS服务器，也就是使用命令指定的DNS查询域名，而不是用/etc/resolv.conf文件中定义的DNS查询。

```
[root@localhost ~]# host www.google.com 8.8.8.8
Using domain server:
Name: 8.8.8.8
Address: 8.8.8.8#53
Aliases:
www.google.com has address 74.125.128.147
www.google.com has address 74.125.128.99
www.google.com has address 74.125.128.106
www.google.com has address 74.125.128.103
www.google.com has address 74.125.128.105
www.google.com has address 74.125.128.104
www.google.com has IPv6 address 2404:6800:4005:c00::93
```

6.4.3 traceroute

在IP包结构中有一个定义数据包生命周期的TTL（Time To Live）字段，该字段用于表明IP数据包的生命值，当IP数据包在网络上传输时，每经过一个路由器该值就减1，当该值减为0时此包就会被路由器丢弃。这种设计可用于避免出现一些由于某种原因始终无法到达目的地的包不断地在互联网上传递（可以形象地称之为“幽灵包”），减少无谓的网络资源耗用。

不过路由器也不是“无声无息”地将TTL值为0的IP包丢弃的，它会同时给发送该IP数据包的主机发送一个ICMP“超时”消息，主机在接收到这个ICMP包后就同时能得到该路由的IP地址。

根据上面两个特点，人们写了一个检测数据包是如何经由路由器的工具——traceroute，我们可以想象一下该工具的工作原理：它先构造出一个TTL值为1的数据包发送给目的主机，这个数据包在经由第一个路由器时，路由器先将TTL值减1变为0，然后将该IP包丢弃，同时给发送一个ICMP消息，这样就得到了经过的第一台路由器的IP地址；然后再构造出一个TTL值为2的数据包，以此类推，就能得到该IP包经历的整条链路的路由器IP。

这里会有一个问题：traceroute如何确认该IP包成功地被目的主机接收了呢？因为目的主机即便收到了TTL值为1的数据包也不会发送ICMP通知给源主机的。这时traceroute所做的工作就是发送一个UDP包给目的主机，同时制定该UDP接收的端口为主机不可能存在的端口，主机在接收到这样的包后，由于端口不可达，则主机会返回一个“端口不可达”的通知，这样就能确认目的主机是否可以接收到数据包。

6.4.4 常见网络故障排查

网络是一切系统赖以正常工作的基础设施，所以保证主机的网络连通性是一切工作得以开展的前提。由于网络协议和设备所具有的复杂性，很多故障解决起来是有难度的，不仅需要工作人员有相应的知识结构来帮助解决问题，有时候还需要他们具有丰富的网络经验。从大多数情况看，网络故障主要分为硬件故障和软件故障两种。

·硬件故障又主要分为网卡物理损坏、链路故障等原因。其中网卡物理损坏是指网卡设备由于使用中发生电子元件损坏而造成网卡设备无法继续使用的情況；链路故障很多时候表现为网线或者水晶头在制作过程中出现线路问题，或由于线路老化等原因造成物理链路断开，从而致使网络无法物理连通的情况。

·软件主要表现为网卡驱动故障，也就是操作系统对网卡驱动的不兼容，这个问题往往需要通过安装对应的网卡设备驱动来解决。

基于以上两点，将解决网络在故障时采用的步骤总结如下（不管其中哪一步中出现问题都需要解决当前的问题才能进行下一步测试，当所有测试都通过了则问题也就解决了）：

第一步是要确认网卡本身是否能正常工作？利用ping工具可以确认这点。输入ping 127.0.0.1，然后看是否能正常ping通？这里的127.0.0.1被称为主机的回环接口，是TCP/IP协议栈正常工作的前提。如果ping不通，一般可以证实为本机TCP/IP协议栈有问题，自然就无法连接网络了。不过，出现这种现象的概率比较低。

第二步是要确认网卡是否出现了物理或驱动故障，使用

ping本机IP地址的方式，如果能ping通则说明本地设备和驱动都正常。

第三步要确认是否能ping通同网段的其他主机。这一步主要是确认二层网络设备（比如交换机或者HUB）工作是否正常。如果ping不通往往说明二层网络上出现了问题，可能涉及交换机的端口工作模式、vlan划分等因素。

第四步要确认是否能ping通网关IP。如果数据包能正常到达网关，则说明主机和本地网络都工作正常。

第五步确认是否能ping通公网上的IP，如果可以则说明本地的路由设置正确，否则就要确认路由设备是否做了正确的nat或路由设置。

第六步确认是否能ping通公网上的某个域名，如果能ping通则说明DNS部分设置正确。

即便实际工作中可能会受到诸如更复杂的网络环境、安全ACL、防火墙等众多因素的影响，而加大了网络排查的困难，但以上步骤是排除网络故障的主要环节，在排除不同的网络之间个性化的设置之后，排查的主要步骤都与此类似。

第7章 进程管理

进程是Linux系统中一个非常重要的概念，但是，这并不意味着我们需要太过接近底层地去了解这些进程是如何运行的、内核是如何管理调度的、时间片是如何轮转分配的等问题，我们所需要关心的是如何控制这些进程，包括查看、启动、关闭、设置优先级等，从而完成好Linux系统工程师的本职工作。

7.1 什么是进程

进程表示程序的一次执行过程，它是应用程序的运行实例，是一个动态的过程。或者可以更简单地描述为：进程是操作系统当前运行的程序。当一个进程开始运行时，就是启动了这个过程。进程包括动态执行的程序和数据两部分。现代操作系统支持多进程处理，这些进程可以接受操作系统的调度，所以说每一个进程都是操作系统进行资源调度和分配的一个独立单位。

所有的进程都可能存在3种状态：运行态、就绪态、阻塞态。运行态表示程序当前实际占用着CPU等资源；就绪态是指程序除CPU之外的一切运行资源都已经就绪，等待操作系统分配CPU资源，只要分配了CPU资源，即可立即运行；而阻塞态是指程序在运行的过程中由于需要请求外部资源（例如I/O资源、打印机等低速的或同一时刻只能独享的资源）而当前无法继续执行，从而主动放弃当前CPU资源转而等待所请求资源。

进程之间又存在互斥和同步的关系。互斥也就是说进程间不能同时运行，必须等待一个进程运行完毕，另一个进程才能运行，比如说不可能有两个进程同时使用同一部打印机打印文件。而进程同步指的是进程间通过某种通信机制实现信息交互。现代计算机使用信号量机制来实现进程间的互斥和同步，它的基本原理是：两个或者多个进程可以通过简单的信号进行合作，一个进程可以被迫在某一位置停止，直到它接收到一个特定的信号。任何复杂的合作需求都可以通过适当的信号结构得到满足。

7.2 进程和程序的区别

要说明进程和程序的区别，首先要了解什么是程序。简单地说，可以将程序看作是对一系列动作执行过程的描述，所以程序只是指令的有序集合，是一个静态的概念。比如说你打开最常用的编辑器，编辑了一段能打印出一些字符的代码，如果你使用的是编译型的语言（比如C语言），对该源代码进行编译连接后，形成的文件就是一个二进制程序。

进程和程序之间既有区别又有天然的联系。进程是动态的，而程序是静态的，进程是程序以及数据在计算机上的一次执行，没有静态的程序也就没有动态的执行。程序是可以以某种形式保存在存储介质上的，而进程只能在运行时存在于计算机的内存中。

以现实生活中的事情来举例，如果说做一件事情需要经过若干既定的步骤，这些步骤可以被写成清单静态地列在纸上，那么它们就是广义上的“程序”，而只有真正开始将计划的步骤付诸实施的过程才是“进程”。

7.3 进程的观察：ps、top

如果想要查看进程，了解当前进程的情况就需要用到相关命令了。其中，ps命令就是一款非常强大的进程查看工具。该命令语法格式如下：

```
[root@localhost ~]# ps
```

参数

#ps

的参数非常多，
在此列出一些常用的参数

#-A

列出所有的进程，和-e
有同样的效果

#-a

列出不和本终端有关的所有进程

#-w

显示加宽可以显示较多信息

#-u

显示有效使用者相关的进程

#aux

显示所有包含其他使用者的进程

#

使用aux

参数的输出：

```
#USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
```

```
#USER:
```

进程拥有者

```
#PID: pid
```

```
##%CPU:
```

占用的CPU

使用率

```
##%MEM:
```

占用的内存使用率

```
#VSZ:
```

占用的虚拟内存大小

```
#RSS:
```

占用的内存大小

```
#TTY:
```

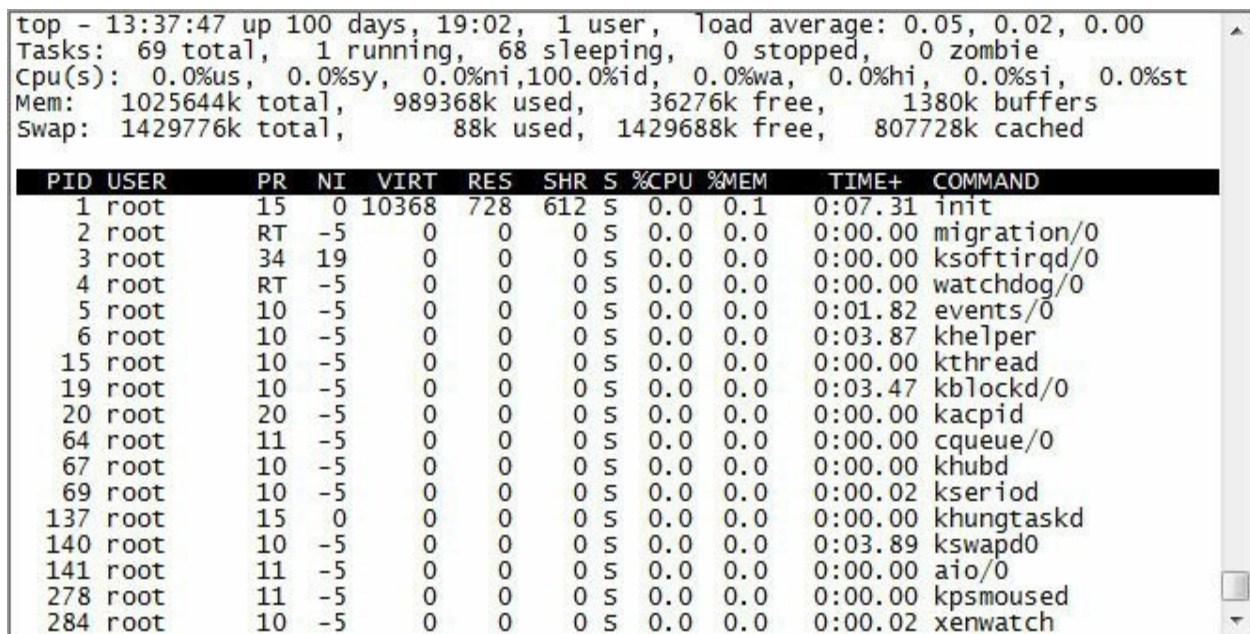
运行的终端的号码

```
#STAT:
```

进程状态：

不可中断	#D:
运行中	#R:
休眠	#S:
暂停	#T:
僵尸进程	#Z:
没有足够的内存可分配	#W:
高优先级的行程	#<:
低优先级的行程	#N:
进程开始时间	#START:
累计使用CPU的时间	#TIME:
执行的命令	#COMMAND:

命令ps输出的只是当前查询状态下进程瞬间的状态信息，如果要想及时动态地查看进程就需要使用top命令了。top命令提供了实时的系统状态监控，可以按照CPU使用、内存使用、执行时间等指标对进程进行排序。图7-1是运行top命令时的输出。



```

top - 13:37:47 up 100 days, 19:02, 1 user, load average: 0.05, 0.02, 0.00
Tasks: 69 total, 1 running, 68 sleeping, 0 stopped, 0 zombie
Cpu(s): 0.0%us, 0.0%sy, 0.0%ni,100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 1025644k total, 989368k used, 36276k free, 1380k buffers
Swap: 1429776k total, 88k used, 1429688k free, 807728k cached

```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1	root	15	0	10368	728	612	S	0.0	0.1	0:07.31	init
2	root	RT	-5	0	0	0	S	0.0	0.0	0:00.00	migration/0
3	root	34	19	0	0	0	S	0.0	0.0	0:00.00	ksoftirqd/0
4	root	RT	-5	0	0	0	S	0.0	0.0	0:00.00	watchdog/0
5	root	10	-5	0	0	0	S	0.0	0.0	0:01.82	events/0
6	root	10	-5	0	0	0	S	0.0	0.0	0:03.87	khelper
15	root	10	-5	0	0	0	S	0.0	0.0	0:00.00	kthread
19	root	10	-5	0	0	0	S	0.0	0.0	0:03.47	kblockd/0
20	root	20	-5	0	0	0	S	0.0	0.0	0:00.00	kacpid
64	root	11	-5	0	0	0	S	0.0	0.0	0:00.00	cqueue/0
67	root	10	-5	0	0	0	S	0.0	0.0	0:00.00	khubd
69	root	10	-5	0	0	0	S	0.0	0.0	0:00.02	kseriod
137	root	15	0	0	0	0	S	0.0	0.0	0:00.00	khungtaskd
140	root	10	-5	0	0	0	S	0.0	0.0	0:03.89	kswapd0
141	root	11	-5	0	0	0	S	0.0	0.0	0:00.00	aio/0
278	root	11	-5	0	0	0	S	0.0	0.0	0:00.00	kpsmoused
284	root	10	-5	0	0	0	S	0.0	0.0	0:00.02	xenwatch

图7-1 运行top命令时的输出

图7-1中的第一行是服务器基础信息，包括top命令的刷新时间为13:37:47，系统已经启动的时间为100天19个小时又两分钟，当前有1个用户登录，系统的负载（load average）为：最近1分钟内的平均系统负载为0.05，最近5分钟内的平均系统负载为0.02，最近15分钟内的平均系统负载为0.00（这说明系统基本是闲置的）。

第二行是当前系统进程概况，一共有69个进程，其中1个正在运行中，68个处于休眠，没有停止的进程，没有僵尸进程。

第三行是CPU信息，us代表用户空间占用的CPU百分比，sy代表内核空间占用的CPU百分比，ni代表改变过优先级的进程占用的CPU百分比，id代表空闲CPU百分比，wa代表I/O等待百分比，hi代表硬中断占用的CPU百分比，si代表软中断占用的CPU百分比。现代计算机一般有多核CPU，要想查看每个逻辑CPU的使用情况，可以在top显示界面中按数字键1。

第四行是物理内存的使用状态，从左到右分别表示物理内

存总量、已使用的内存、空闲内存、缓存使用的内存。

第五行是虚拟内存的使用状态，其中，前三列和物理内存的意义一致，最后一个代表缓冲的交换区总量。

再往下的所有信息就是动态的进程信息了，表7-1中给出了这部分每一列的含义。

进程信息区中的信息只是top默认显示的11个字段，如果要显示更多的字段，可以在top显示界面中按字母键f。按该键后，前面打了*号的就是当前显示的字段，要想显示更多的字段可以按一下字段前面的字母对应的键。比如，本例中按了b和c键，选中后按回车键即可返回top显示界面，如图7-2所示。

另外，默认情况下top显示的进程是按照CPU使用率来进行排序的，如果要另选排序规则怎么办呢？可以按大写字母O键进入排序选择页，然后按一下字段前面的字母对应的键来选择排序字段，之后按回车键返回即可，如图7-3所示。

表7-1 top命令动态进程信息中每列的含义

字段	含 义
PID	进程 id
USER	进程所有者
PR	进程优先级
NI	nice 值，负值表示高优先级，正值表示低优先级
VIRT	进程使用的虚拟内存总量，单位为 Kb，VIRT=SWAP+RES
RES	进程使用的未被换出的物理内存大小，单位为 Kb，RES=CODE+DATA
SHR	共享内存大小，单位为 Kb
%CPU	上次更新到现在的 CPU 时间占用百分比
%MEM	进程使用的物理内存百分比
TIME+	进程使用的 CPU 时间总计，单位为 1/100 秒
COMMAND	进程名称（命令名 / 命令行）

```

Current Fields: AEHIOQTWKNMBCdfgjplrsuvyZX for window 1:Def
Toggle fields via field letter, type any other key to return

* A: PID          = Process Id          u: nFLT          = Page Fault count
* E: USER         = User Name           v: nDRT          = Dirty Pages count
* H: PR          = Priority              y: WCHAN         = Sleeping in Function
* I: NI          = Nice value           z: Flags         = Task Flags <sched.h>
* O: VIRT        = Virtual Image (kb)   * X: COMMAND     = Command name/line
* Q: RES         = Resident size (kb)
* T: SHR        = Shared Mem size (kb)
* W: S          = Process Status
* K: %CPU       = CPU usage
* N: %MEM       = Memory usage (RES)
* M: TIME+     = CPU Time, hundredths
* B: PPID      = Parent Process Pid
* C: RUSER     = Real user name
d: UID        = User Id
f: GROUP      = Group Name
g: TTY        = Controlling Tty
j: P          = Last used cpu (SMP)
p: SWAP       = Swapped size (kb)
l: TIME       = CPU Time
r: CODE       = Code size (kb)
s: DATA      = Data+Stack size (kb)

Flags field:
0x00000001 PF_ALIGNWARN
0x00000002 PF_STARTING
0x00000004 PF_EXITING
0x00000040 PF_FORKNOEXEC
0x00000100 PF_SUPERPRIV
0x00000200 PF_DUMPCORE
0x00000400 PF_SIGNALED
0x00000800 PF_MEMALLOC
0x00002000 PF_FREE_PAGES (2.5)
0x00008000 debug flag (2.5)
0x00024000 special threads (2.5)
0x001D0000 special states (2.5)
0x00100000 PF_USEDFP (thru 2.4)

```

图7-2 显示更多动态字段

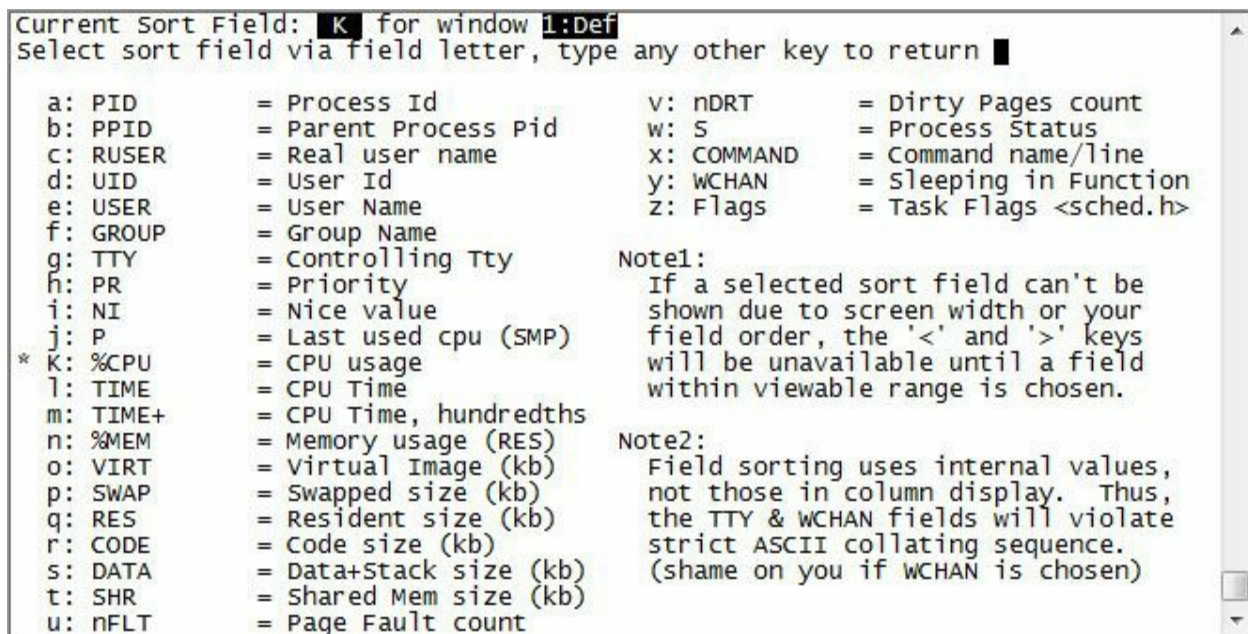


图7-3 选择排序方式

在top显示页面中还有一些快捷键可以使用，比如按字母P键表示按照CPU的使用率排序，按字母M键表示按照Memory的使用率排序，按字母N键表示以PID排序，按字母T键表示按照CPU使用时间排序，按字母K键则表示kill进程，按字母R键表示可以renice一个进程等。注意快捷键是区分大小写的。更多可用的方式可以按问号（?）键进入帮助模式。

7.4 进程的终止：kill、killall

要终止一个进程，需要通过kill、killall等命令来实现。比如说有部分进程由于某种原因已经死掉或者工作异常，或者要停止一些非关键或非数据业务的进程，那么这时就需要使用这些命令来终止进程。这些命令的原理都是向内核发送一个系统操作信号以及某个进程的标识号，使得内核对指定标识号的进程进行相应的操作。

一般来说，kill命令需要和ps命令联合使用。原因是kill后面跟的应该是需要被终止的进程的PID。典型用法是使用ps查出进程的PID，然后使用kill将其终止。kill的使用方法如下：

```
[root@localhost ~]# kill [
信号代码]
进程ID
```

假设系统中的dhcpd进程由于某种原因需要终止，那么首先要查找到该进程的PID（从下面的输出中可以看到该PID为2877），然后kill这个PID。完成这个操作后再看dhcpd进程，就已经不在了。

```
[root@localhost ~]# ps -ef | grep dhcp
root      2877      1  0 18:59 ?                00:00:00 /usr/sbin/dhc
#
这里找出dhcpd
的PID
是2877
#
有个更快速的方式来寻找进程的PID
，即使用pidof
命令
#[root@localhost ~]# pidof dhcpd
#2877
```

```
[root@localhost ~]# kill 2877
```

命令kill后可以跟的信号代码一共有64种，使用kill-l就可以看到具体有哪些，如图7-4所示。但是常用的一般只有3个，即HUP（1）、KILL（9）、TERM（15），分别代表重启、强行杀掉、正常结束。

```
[root@localhost ~]# kill -l
1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL
5) SIGTRAP     6) SIGABRT     7) SIGBUS      8) SIGFPE
9) SIGKILL     10) SIGUSR1    11) SIGSEGV    12) SIGUSR2
13) SIGPIPE    14) SIGALRM    15) SIGTERM    16) SIGSTKFLT
17) SIGCHLD    18) SIGCONT    19) SIGSTOP    20) SIGTSTP
21) SIGTTIN    22) SIGTTOU    23) SIGURG     24) SIGXCPU
25) SIGXFSZ    26) SIGVTALRM 27) SIGPROF    28) SIGWINCH
29) SIGIO      30) SIGPWR     31) SIGSYS     34) SIGRTMIN
35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3 38) SIGRTMIN+4
39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
[root@localhost ~]#
```

图7-4 kill命令可用的信号代码

信号1代表重启，假设需要重启系统中的httpd服务，先查主httpd进程的PID号，这里为2935，如图7-5所示（注意，在图7-5中，第一次查询的时候，发现有若干个httpd进程，但是主进程只有一个，即由root启动的、PID为2935的第一个进程，其他的都是该进程的子进程）。使用kill-1 2935后，再查看httpd进程的时候，发现主进程的PID没有变化，而子进程的PID都在同一时刻发生了变化，这说明主进程确实经过了重启。这也说明，使用kill-1重启进程的时候实际上是不会改变主进程的PID的，也就是说只是发生了原地重启，或者说“软重启”。

```
[root@localhost ~]# ps -ef | grep httpd | grep -v grep
root      2935      1  0 18:59 ?        00:00:00 /usr/sbin/httpd
apache    5208     2935  0 21:14 ?        00:00:00 /usr/sbin/httpd
apache    5209     2935  0 21:14 ?        00:00:00 /usr/sbin/httpd
apache    5210     2935  0 21:14 ?        00:00:00 /usr/sbin/httpd
apache    5211     2935  0 21:14 ?        00:00:00 /usr/sbin/httpd
apache    5212     2935  0 21:14 ?        00:00:00 /usr/sbin/httpd
apache    5213     2935  0 21:14 ?        00:00:00 /usr/sbin/httpd
apache    5214     2935  0 21:14 ?        00:00:00 /usr/sbin/httpd
apache    5215     2935  0 21:14 ?        00:00:00 /usr/sbin/httpd
[root@localhost ~]# kill -1 2935
[root@localhost ~]# ps -ef | grep httpd | grep -v grep
root      2935      1  0 18:59 ?        00:00:00 /usr/sbin/httpd
apache    5238     2935  0 21:18 ?        00:00:00 /usr/sbin/httpd
apache    5239     2935  0 21:18 ?        00:00:00 /usr/sbin/httpd
apache    5240     2935  0 21:18 ?        00:00:00 /usr/sbin/httpd
apache    5241     2935  0 21:18 ?        00:00:00 /usr/sbin/httpd
apache    5242     2935  0 21:18 ?        00:00:00 /usr/sbin/httpd
apache    5243     2935  0 21:18 ?        00:00:00 /usr/sbin/httpd
apache    5244     2935  0 21:18 ?        00:00:00 /usr/sbin/httpd
apache    5245     2935  0 21:18 ?        00:00:00 /usr/sbin/httpd
[root@localhost ~]#
```

图7-5 使用kill重启进程

前面成功地使用不带信号代码的kill停止了dhcpd进程，但实际上有一些进程因为运行中出现问题而无法通过这种方式停止，在这种情况下就需要使用-9参数强行停止该进程了，其效果是立即杀死进程，而且该信号无法被阻塞或忽略。但是这个命令也有其天然的危险，就是进程可能会直接被系统终止，而没有清理之前申请的内存，这会造成一定程度的“内存泄露”，因此一般情况下不建议使用。而-15这个参数就比较温和了，它会使进程正常退出，它也是Linux默认的程序中断信号（也就是在不加参数的情况下默认使用的信号）。

由于使用kill命令时要先查询到想要终止的进程的PID，也就是说操作对象是数字，相对来说会比较麻烦，而且在实际的工作中，如果看错了PID其后果是无法估计的（想象一下，如果看错或是输错了PID，恰巧将一个非常重要的应用程序给kill了，那就无异于一场灾难）。事实上，想要终止进程时还有第二个命令可以选择，即killall命令，它可以直接使用进程的名字而不是PID，如果要停止系统中所有的httpd进程，那么只要按照以下方法操作就可以了：

```
[root@localhost ~]# killall httpd
```

这个命令不但简单而且更为安全。

7.5 查询进程打开的文件：lsof

lsof (list open files) 是一个列出当前系统中所有打开文件的工具。早在第1章中就提到过，Linux中一切皆文件，所以在系统中，被打开的文件可以是普通文件、目录、网络文件系统中的文件、字符设备、管道、socket等。那么如何知晓现在系统打开的是哪些文件呢，这时**lsof**命令就有用武之地了。不过，这个命令在系统中可能并未默认安装，在CentOS下如果可以联网，简单输入**yum install lsof-y**即可安装该工具，如果是在RedHat下，则需要到原始安装光盘中寻找**lsof**的rpm包进行安装。关于如何安装软件包，将在下一章中详细描述。该命令的使用方法如下：

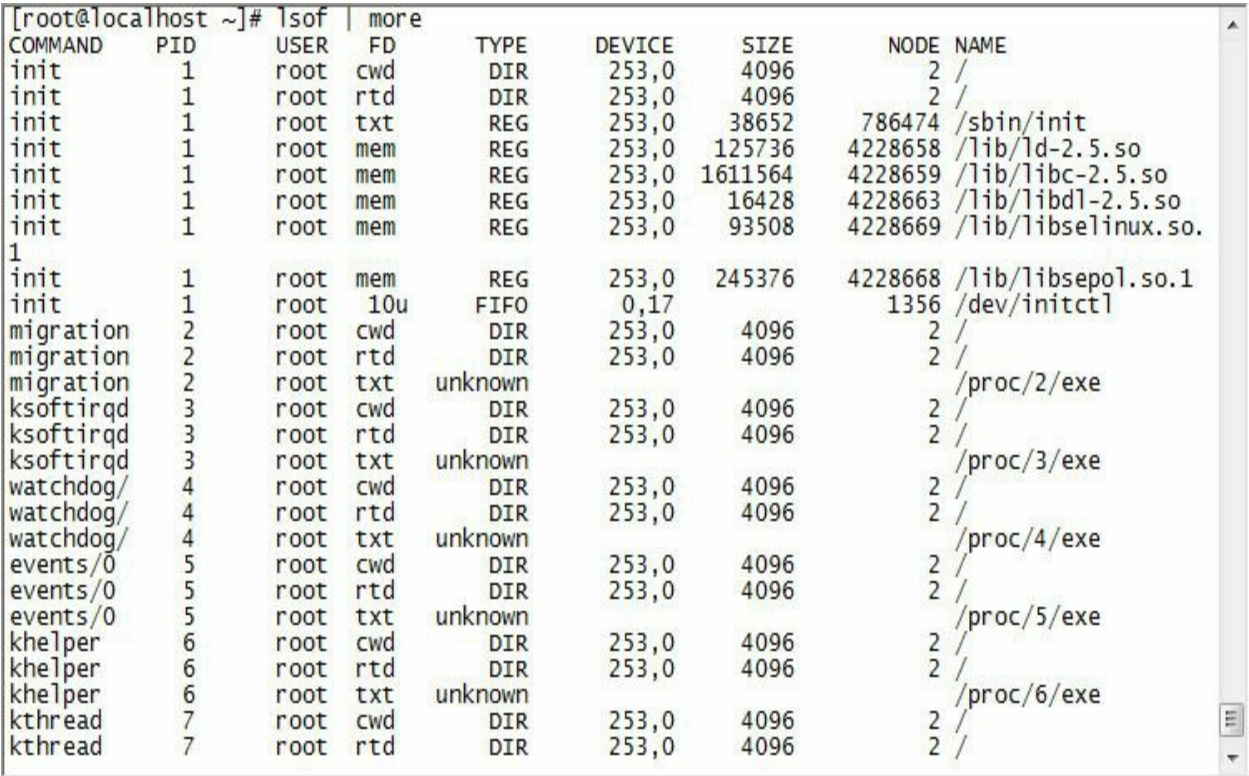
```
[root@localhost ~]# lsof
[options
] filename
#
常用的参数列表
#lsof filename
显示打开指定文件的所有进程
#lsof -c string
显示COMMAND
列中包含指定字符的进程所有打开的文件
#lsof -u username
显示所属于user
进程打开的文件
#lsof -g gid
显示归属于gid
的进程情况
#lsof +d /DIR/
显示目录下被进程打开的文件
#lsof +D /DIR/
同上，但是会搜索目录下的所有目录，时间相对较长
#lsof -d FD
显示指定文件描述符的进程
#lsof -n
不将IP
转换为hostname
```

，默认是不加-n
参数
#lsof -i
用以显示符合条件的进程情况
#lsof -i[46] [protocol][@hostname|hostaddr][:service|port]
46
指IPv4
或IPv6
protocol
指TCP
或UDP
hostname
指主机名
hostaddr
是IPv4
地址
service
是/etc/service
中的service name
port
是端口号

这个命令可以在不加任何参数的情况下直接运行，但是该命令一定需要用root账号来执行，因为lsof在运行时需要访问很多核心文件，需要的权限很高，其所输出的是目前系统中所有打开的文件，如图7-6所示。输出的字段有COMMAND、PID、USER、FD、TYPE、DEVICE、SIZE、NODE、NAME9列，这9个字段的意思如下：

- COMMAND：进程的名称。
- PID：进程标识符。
- USER：进程所有者。
- FD：文件描述符，应用程序通过文件描述符识别该文件。
- TYPE：文件类型，如DIR、REG等。

- DEVICE：磁盘的名称。
- SIZE：文件大小。
- NODE：索引节点。
- NAME：打开文件的全路径名称。



COMMAND	PID	USER	FD	TYPE	DEVICE	SIZE	NODE	NAME
init	1	root	cwd	DIR	253,0	4096	2	/
init	1	root	rtd	DIR	253,0	4096	2	/
init	1	root	txt	REG	253,0	38652	786474	/sbin/init
init	1	root	mem	REG	253,0	125736	4228658	/lib/ld-2.5.so
init	1	root	mem	REG	253,0	1611564	4228659	/lib/libc-2.5.so
init	1	root	mem	REG	253,0	16428	4228663	/lib/libdl-2.5.so
init	1	root	mem	REG	253,0	93508	4228669	/lib/libselinux.so.
1								
init	1	root	mem	REG	253,0	245376	4228668	/lib/libsepol.so.1
init	1	root	10u	FIFO	0,17		1356	/dev/initctl
migration	2	root	cwd	DIR	253,0	4096	2	/
migration	2	root	rtd	DIR	253,0	4096	2	/
migration	2	root	txt	unknown				/proc/2/exe
ksoftirqd	3	root	cwd	DIR	253,0	4096	2	/
ksoftirqd	3	root	rtd	DIR	253,0	4096	2	/
ksoftirqd	3	root	txt	unknown				/proc/3/exe
watchdog/	4	root	cwd	DIR	253,0	4096	2	/
watchdog/	4	root	rtd	DIR	253,0	4096	2	/
watchdog/	4	root	txt	unknown				/proc/4/exe
events/0	5	root	cwd	DIR	253,0	4096	2	/
events/0	5	root	rtd	DIR	253,0	4096	2	/
events/0	5	root	txt	unknown				/proc/5/exe
khelper	6	root	cwd	DIR	253,0	4096	2	/
khelper	6	root	rtd	DIR	253,0	4096	2	/
khelper	6	root	txt	unknown				/proc/6/exe
kthread	7	root	cwd	DIR	253,0	4096	2	/
kthread	7	root	rtd	DIR	253,0	4096	2	/

图7-6 lsdf的输出

Linux系统中有很多日志文件会不断地被写入、更新，/var/log/messages就是其中的一个。现在来看一下当前有什么进程正在使用该文件（syslogd是系统中负责写系统日志的进程）。

```
[root@localhost ~]# lsdf /var/log/messages
```

COMMAND	PID	USER	FD	TYPE	DEVICE	SIZE	NODE	NAME
syslogd	2428	root	1w	REG	253,0	109860	4161767	/var/log/messa

在第4章中曾经将新创建的磁盘挂载到/root/newDisk下，现在假设要进入该目录，然后尝试将其umount卸载，系统提示device is busy，无法卸载。这时候使用lsof命令确认了一下，确实有进程在占用这个目录，于是通过cd命令找到家目录，然后再使用lsof确认，发现已经没有进程占用，这时候再umount就不会报错了。具体如下所示：

```
[root@localhost ~]# cd /root/newDisk/
[root@localhost newDisk]# umount /root/newDisk/
umount: /root/newDisk: device is busy
umount: /root/newDisk: device is busy
[root@localhost newDisk]# lsof /root/newDisk/
COMMAND  PID USER  FD   TYPE DEVICE SIZE NODE NAME
bash     3310 root   cwd   DIR   8,17 4096    2 /root/newDisk/
lsof     3971 root   cwd   DIR   8,17 4096    2 /root/newDisk/
lsof     3972 root   cwd   DIR   8,17 4096    2 /root/newDisk/
[root@localhost newDisk]# cd
[root@localhost ~]# lsof /root/newDisk/
[root@localhost ~]# umount /root/newDisk/
```

使用lsof还可以查找使用了某个端口的进程，比如说如果系统中运行了sshd进程（基本上都是默认运行的），则该进程默认会绑定22端口，让我们来确认一下：

```
[root@localhost ~]# lsof -i:22
COMMAND  PID USER  FD   TYPE DEVICE SIZE NODE NAME
sshd     2815 root   3u   IPv6  9402      TCP *:ssh (LISTEN)
```

如果你发现系统中打开了一些未知的端口，可以使用这个方法确认具体是什么进程在使用。

使用lsof命令还有个更实用的功能，就是可以通过其恢复被删除的文件——但这是有条件的，必须是文件正在被某个进程使用，而且该进程未停止（也就是依然拥有打开文件的句

柄)。我们知道，在Windows下恢复数据相对来说比较简单，因为Windows提供了回收站功能，删除的文件可以在其中简单地被找回。另外，Windows系统下的第三方软件也是非常丰富的，即使回收站被清空了，也有可能使用这些第三方软件协助恢复数据的软件。

现假设文件/var/log/messages不小心被删除了，首先来确认一下当前是否有进程正在使用这个文件，如果有则可以继续，如果没有就无法使用该方法继续了。本例中看到有个PID为2449的进程正在使用该文件，那么接下来只要找到对应/proc目录下的文件就可以了。具体看以下的命令演示：

```
[root@localhost ~]# lsof | grep message
syslogd    2449      root    1w      REG      253,0    149423
[root@localhost ~]# cat /proc/2449/fd/2 > /var/log/messages
[root@localhost ~]# Service syslogd restart
```

7.6 进程优先级调整：nice、renice

在学习top时，我们看到其输出中有NI字段，标记了对应进程的优先级，该字段的取值范围是-20~19，数值越低代表优先级越高，也就能更多地被操作系统调度运行，如果一个进程在启动时并没有设定nice优先级，则默认使用0。普通用户也可以给自己的进程设定nice优先级，但是范围只限于0~19。不过top中不是还有一个PR字段吗，它也是进程的“优先级”，这两个概念怎么理解呢？实际上，Linux使用了“动态优先级”的调度算法来确定每一个进程的优先级，一个进程的最终优先级=优先级+nice优先级。

nice命令仅限于在启动一个进程的时候同时赋予其nice优先级，比如你自己写了一个脚本job.sh，你想以比较高的优先级来运行它，就可以这么做：

```
[root@localhost ~]# nice -n -10 ./job.sh
```

对于已经启动的进程，可以用renice命令进行修改，不过，这需要先查询出该进程的PID（使用ps命令）。假设现在需要将PID为5555的进程的nice优先级调整为-10，则可以这么做：

```
[root@localhost ~]# renice -10 -p 5555
```

除了使用renice外，还可以使用top提供的功能来修改，前提也是要查到该进程的PID，然后在top界面中按r键，在出现的PID to renice后输入PID，如图7-7所示。然后在出现的renice PID***to value后输入修改后的nice优先级既可，如图7-8所

示。

```
top - 20:45:17 up 5:21, 1 user, load average: 0.00, 0.00, 0.00
Tasks: 86 total, 1 running, 85 sleeping, 0 stopped, 0 zombie
Cpu(s): 0.1%us, 0.1%sy, 0.1%ni, 99.4%id, 0.0%wa, 0.0%hi, 0.3%si, 0.0%st
Mem: 2075468k total, 243404k used, 1832064k free, 39528k buffers
Swap: 2097144k total, 0k used, 2097144k free, 154312k cached
PID to renice:
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1	root	15	0	2072	636	544	S	0.0	0.0	0:00.93	init
2	root	RT	-5	0	0	0	S	0.0	0.0	0:00.00	migration/0
3	root	34	19	0	0	0	S	0.0	0.0	0:00.00	ksoftirqd/0
4	root	RT	-5	0	0	0	S	0.0	0.0	0:00.00	watchdog/0
5	root	10	-5	0	0	0	S	0.0	0.0	0:00.04	events/0
6	root	10	-5	0	0	0	S	0.0	0.0	0:00.00	khelper
7	root	11	-5	0	0	0	S	0.0	0.0	0:00.00	kthread
10	root	18	-5	0	0	0	S	0.0	0.0	0:00.14	kblockd/0

图7-7 使用top修改进程的优先级

```
top - 20:43:20 up 5:19, 1 user, load average: 0.00, 0.00, 0.00
Tasks: 86 total, 1 running, 85 sleeping, 0 stopped, 0 zombie
Cpu(s): 0.1%us, 0.1%sy, 0.1%ni, 99.4%id, 0.0%wa, 0.0%hi, 0.3%si, 0.0%st
Mem: 2075468k total, 243404k used, 1832064k free, 39496k buffers
Swap: 2097144k total, 0k used, 2097144k free, 154312k cached
Renice PID 516 to value:
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1	root	15	0	2072	636	544	S	0.0	0.0	0:00.93	init
2	root	RT	-5	0	0	0	S	0.0	0.0	0:00.00	migration/0
3	root	34	19	0	0	0	S	0.0	0.0	0:00.00	ksoftirqd/0
4	root	RT	-5	0	0	0	S	0.0	0.0	0:00.00	watchdog/0
5	root	10	-5	0	0	0	S	0.0	0.0	0:00.04	events/0
6	root	10	-5	0	0	0	S	0.0	0.0	0:00.00	khelper
7	root	11	-5	0	0	0	S	0.0	0.0	0:00.00	kthread
10	root	12	-5	0	0	0	S	0.0	0.0	0:00.14	kblockd/0

图7-8 输入修改后的nice优先级

第8章 Linux下的软件安装

8.1 源码包编译安装

由于计算机无法直接执行用高级语言编写的源程序，因此要想运行程序，就需要使用一种机制来让计算机识别，这样程序才可能运行起来。一般来说，计算机中存在解释型和编译型两种语言。所谓解释型语言，就是计算机逐条取出源码文件的指令，将其转化成机器指令，并执行这个指令的过程。而编译型语言是指在程序运行前就将所有的源代码一次性转化为机器代码（一般为二进制程序），再运行这个过程。本节将演示如何在Linux下使用源码包安装软件。

8.1.1 编译、安装、打印HelloWorld程序

本节将带领大家完成从源代码编写到源码编译，再到程序安装的过程，希望通过该过程的学习，能让大家了解源码编译安装的原理。由于几乎所有的开源程序使用的都是C语言，所以这里也使用C语言来演示如何编写、编译、安装一个打印“Hello,world!”程序。

首先，根据软件需求写出源代码（本例中只要求能打印出HelloWorld）。可使用vi编译器编写HelloWorld.c文件，方式如下（常见编辑器包括vi编辑器的用法下一章中将具体讲）：

```
[root@localhost ~]# vi HelloWorld.c  #
回车
#
这里将进入vi
命令模式，按i
键进入编辑模式，输入如下内容
#
输入完成后，按Esc
键，然后输入冒号，再按x
键并按回车键
#include <stdio.h>
int main(void) {
printf("Hello,world!\n");
return 0;
}
```

有了HelloWorld.c这个源码文件，下面就需要使用gcc工具将该源代码编译成一个可执行的二进制程序了。如果你目前使用的Linux完全是按照第1章演示的安装过程安装的，那么很有可能系统中并没有gcc命令，要想安装gcc请参考下一节中的“使用rpm包安装gcc”；如果确认系统中存在该命令，则可以使用如下命令将源代码编译为可执行的二进制文件：

```
[root@localhost ~]# gcc HelloWorld.c -o HelloWorld
#
如果没有gcc
命令，将会出现如下报错信息，否则将生成文件HelloWorld
#-bash: gcc: command not found
[root@localhost ~]# ls HelloWorld      #
得到了HelloWorld
二进制文件
HelloWorld
```

编译完成后，我们得到了二进制可执行文件HelloWorld。那么接下来是否可以直接运行这个命令呢？答案是否定的，尝试运行该命令的时候，系统给出了command not found的报错信息，如下所示：

```
[root@localhost ~]# HelloWorld
-bash: HelloWorld: command not found
```

系统中确实已经有了HelloWorld这个程序，可为什么还是说不存在这个命令呢？这就不得不说到系统变量PATH了，它被称作为Linux系统的“环境变量”，可使用如下命令查看当前PATH变量定义的内容：

```
[root@localhost ~]# echo $PATH
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/root/bin
```

可以看到，PATH变量中是一些由冒号隔开的路径，当输入一个命令时，系统会到PATH所定义的路径中去寻找该命令，找到后就会执行该命令。也就是说，本例中在输入命令HelloWorld并按回车键时，系统先从目录/usr/kerberos/sbin中寻找是否有这个文件，如果找不到就继续从目录/usr/kerberos/bin中寻找，再找不到就到目录/usr/local/sbin中找，以此类推。如

果在PATH定义的所有目录中都找不到该文件，这时系统就会提示command not found。

想象一下，如果不使用这种机制，那么运行任何命令都需要键入某个命令的全路径，将非常麻烦。还记得在第3章中学习过的which命令吗？它的工作原理也是到环境变量PATH中寻找某个命令，事实上如果使用which找不到某个命令，则说明该命令由于找不到而无法被执行：

```
[root@localhost ~]# which HelloWorld
/usr/bin/which: no HelloWorld in (/usr/kerberos/sbin:/usr/kerbin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/
```

由于程序HelloWorld当前所在的路径是/root/HelloWorld，它并不存在于当前PATH中，所以这里的报错是正常的。解决这里出现的command not found错误有三种方法，第一种方法是在/root目录中使用./HelloWorld执行该命令，或者引用该命令的全路径来执行；第二种方法是将HelloWorld复制到任意一个当前PATH变量包含的目录中；第三种方法是将/root目录追加到PATH变量中。以上方法任意选择使用一种即可，如下所示：

```
#
第一种方法
#
使用 ./
执行或使用全路径执行
[root@localhost ~]# pwd
/root
[root@localhost ~]# ./HelloWorld
Hello, world!
[root@localhost ~]# /root/HelloWorld
Hello, world!
#
第二种方法
```

```
#
将HelloWorld
复制到任一PATH
变量包含的目录中，这里使用/bin
目录
[root@localhost ~]# cp HelloWorld /bin/
[root@localhost ~]# which HelloWorld
/bin/HelloWorld
[root@localhost ~]# HelloWorld
Hello, world!
#
第三种方法
#
将/root
目录追加到PATH
变量中，注意看追加目录的方法
#
在尝试使用该方法之前，如果已经使用了第二种方法，则先删除之前复制的文件
#[root@localhost ~]# rm /bin/HelloWorld
#rm: remove regular file `/bin/HelloWorld'? y
[root@localhost ~]# export PATH=$PATH:/root
[root@localhost ~]# which HelloWorld
/root/HelloWorld
[root@localhost ~]# HelloWorld
Hello, world!
```

此处需要注意的是，虽然第三种方法和第二种方法的原理是一致的，但是第三种方法一般在重启主机或重新登录之后就失效了，原因是这种方法并没有将所定义的环境变量保存到任何配置文件中。在这种方法下，可以使用以下命令保存变量PATH的值：

```
echo "export PATH=$PATH:/root" >> /etc/rc.local
```

以上就是源码编译安装软件的原理，简单总结一下就是编写源代码→编译源码生成二进制可执行性文件（也就是程序）→复制该文件到任一PATH变量包含的目录中。

本例中用于演示的软件功能十分简单，只要能打

印“Hello,world!”就可以了，所以只需要一个单独的源码文件就可以搞定，而且代码也非常简单。在实际工作中，软件的需求往往比较复杂，而且大多是基于模块化开发的思想来实现的，所以一个软件往往需要多个源码文件和各类配置文件，在编译的时候也需要严格按照一定的过程进行编译，比如说需要先编译出某些模块文件之后，才能最终编译并生成主程序。而这个过程也只有软件开发者自己才清楚，这意味着只有在软件开发者提供了详细的编译步骤文档的前提下，拿到该源码包的人才能按照其规定的编译顺序来编译生成程序。在这种情况下，为了方便软件安装，可以使用Makefile简化整个过程，由于本书并不涉及C语言开发以及Makefile的语法，所以这里并不打算深入讲解Makefile，只做一些演示。在/root目录中编辑Makefile，内容如下所示：

```
[root@localhost ~]# cat Makefile
HelloWorld:HelloWorld.o
        gcc -o HelloWorld HelloWorld.c #
前面不是空格，而是一个Tab
install:
        cp HelloWorld /bin/ #
此处前面的也是一个Tab
```

有了Makefile之后，编译安装HelloWorld程序就变得更简单了，只需要以下两条命令即可：

```
#
第一步，输入make
命令，这会完成编译的过程
[root@localhost ~]# make
gcc -o HelloWorld HelloWorld.c #
这里是make
命令执行后的输出
#
第二步，输入make install
命令，这会完成软件的复制
```

```
[root@localhost ~]# make install
cp HelloWorld /bin/ #
注意这里是make install
的输出，不需要人工复制
#
然后就可以直接执行命令了
[root@localhost ~]# HelloWorld
Hello, world!
```

事实上，有很多开源软件自身是不包含Makefile的，特别是在模块化程度较高的软件中，都不包含Makefile，而需要用户根据具体的需求使用软件包目录中的configure工具，生成适合用户特定需求的Makefile，所以典型的源码编译安装软件的过程包括以下3步：

第一步，运行configure命令，并结合必要的参数以生成Makefile；

第二步，运行make命令生成各类模块和主程序；

第三步，运行make install命令将必要的文件复制到安装目录中。

以上3步都需要在对应软件包目录的根目录下运行。

8.1.2 使用源码包编译安装Apache

Apache是Apache软件基金会的一个开源Web服务器。Apache的前身是NCSAhttpd，当该项目停顿后，原先使用该服务器软件的人们架设了一个论坛用以交换各自开发的补丁程序，在这个软件被开源后还不断有人为它开发新功能、新特性，以及发布修正bug的补丁，大家笑谈它其实就是一个充满补丁的软件，所以称其为“a patchy server”，后简称Apache。由于其具有良好的安全性和跨平台（几乎可以运行在各类操作系统平台之上），因此被广泛使用，并成为最流行的Web服务器软件之一。事实上，Apache至今依然是世界上使用比例最高的Web服务器（市场占有率一度达到60%），有很多著名的网站都在使用它。它的特点是简单、快速、稳定，支持SSL加密、虚拟主机等功能。同时由于其模块化程度非常高，使得它极易进行功能扩展。

经过上一节的铺垫，我们已经了解了编译安装软件的原理以及一般必要的步骤。本节将演示如何编译安装Apache，希望读者能跟着动手实践，增强对编译安装软件的理解。

首先下载Apache的源码包，下载地址为：

`http://mirrors.cnnic.cn/apache/httpd/httpd-2.2.23.tar.gz`

当前Apache最稳定的大版本为2.2，如果想使用不同的版本，可到Apache的官方主页<http://www.apache.org>下载。

在Linux系统中，一般在/usr/local/src/目录里下载源码包（这不是硬规定，而是一个良好的习惯），进入该目录后可使用wget `http://mirrors.cnnic.cn/apache/httpd/httpd-2.2.23.tar.gz`命令下载，如图8-1所示。

```
[root@localhost local]# cd /usr/local/src/
[root@localhost src]# wget http://mirrors.cnnic.cn/apache/httpd/httpd-2.2.23.tar.gz
--2013-02-14 17:07:42-- http://mirrors.cnnic.cn/apache/httpd/httpd-2.2.23.tar.gz
Resolving mirrors.cnnic.cn... 123.125.244.87
Connecting to mirrors.cnnic.cn|123.125.244.87|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 7374712 (7.0M) [application/x-gzip]
Saving to: `httpd-2.2.23.tar.gz'

13% [====>] 986,653 85.7K/s eta 92s
```

图8-1 下载Apache源码包

下载完成后把源码包解压出来，并进入/usr/local/src/httpd-2.2.23目录，如图8-2所示。

```
[root@localhost src]# tar zxvf httpd-2.2.23.tar.gz && cd httpd-2.2.23
httpd-2.2.23/
httpd-2.2.23/emacs-style
httpd-2.2.23/httpd.dsp
httpd-2.2.23/libhttpd.dsp
httpd-2.2.23/.deps
httpd-2.2.23/Makefile.in
httpd-2.2.23/include/
httpd-2.2.23/include/scoreboard.h
httpd-2.2.23/include/ap_regkey.h
httpd-2.2.23/include/ap_compat.h
httpd-2.2.23/include/http_config.h
httpd-2.2.23/include/util_time.h
httpd-2.2.23/include/ap_mmn.h
httpd-2.2.23/include/ap_provider.h
```

图8-2 解压文件并进入安装目录

进入目录后，需要使用configure工具生成Makefile，运行configure的方式如下：

```
[root@localhost httpd-2.2.23]# ./configure --
参数1 --
参数2...
```

由于配置Apache时可以加入的参数非常多，而且对于新手来说也确实很难搞明白那么多参数各自的意义（具体的可用参数可以在/usr/local/src/httpd-2.2.23/configure中看到），因此这里介绍两个比较简单的参数来完成配置。第一个参数是--prefix=/usr/local/apache/，用于指定安装路径，一般来说建议自

行编译安装的软件放置的目录为/usr/local/；第二个参数是--enable-modules=most，用于启用Apache的绝大部分模块，非常适合新手使用。在按回车键后configure会产生大量的输出，包括检查编译环境（是否有gcc工具以及软件依赖关系）等，中间出现任何错误都会导致配置失败（会有error报错并中断配置过程）。如果一切顺利，将会在当前目录下生成Makefile文件，如图8-3所示。然后使用make和make install安装即可。此处也会产生大量输出，如图8-4所示。完成后将会出现/usr/local/apache目录。

```
[root@localhost httpd-2.2.23]# ./configure --prefix=/usr/local/apache/ --enable-modules=most
checking for chosen layout... Apache
checking for working mkdir -p... yes
checking build system type... i686-pc-linux-gnu
checking host system type... i686-pc-linux-gnu
checking target system type... i686-pc-linux-gnu

Configuring Apache Portable Runtime library ...
```

图8-3 配置Apache编译参数

```
[root@localhost httpd-2.2.23]# make && make install
Making all in srclib
make[1]: Entering directory `/usr/local/src/httpd-2.2.23/srclib'
Making all in apr
make[2]: Entering directory `/usr/local/src/httpd-2.2.23/srclib/apr'
make[3]: Entering directory `/usr/local/src/httpd-2.2.23/srclib/apr'
/bin/sh /usr/local/src/httpd-2.2.23/srclib/apr/libtool --silent --mode=compile gcc -g -O2 -pth
read -DHAVE_CONFIG_H -DLINUX=2 -D_REENTRANT -D_GNU_SOURCE -D_LARGEFILE64_SOURCE -I./includ
e -I/usr/local/src/httpd-2.2.23/srclib/apr/include/arch/unix -I./include/arch/unix -I/usr/loca
l/src/httpd-2.2.23/srclib/apr/include/arch/unix -I/usr/local/src/httpd-2.2.23/srclib/apr/inclu
de -o passwd/apr_getpass.lo -c passwd/apr_getpass.c && touch passwd/apr_getpass.lo
/bin/sh /usr/local/src/httpd-2.2.23/srclib/apr/libtool --silent --mode=compile gcc -g -O2 -pth
read -DHAVE_CONFIG_H -DLINUX=2 -D_REENTRANT -D_GNU_SOURCE -D_LARGEFILE64_SOURCE -I./includ
e -I/usr/local/src/httpd-2.2.23/srclib/apr/include/arch/unix -I./include/arch/unix -I/usr/loca
l/src/httpd-2.2.23/srclib/apr/include/arch/unix -I/usr/local/src/httpd-2.2.23/srclib/apr/inclu
de -o strings/apr_cpysrtn.lo -c strings/apr_cpysrtn.c && touch strings/apr_cpysrtn.lo
/bin/sh /usr/local/src/httpd-2.2.23/srclib/apr/libtool --silent --mode=compile gcc -g -O2 -pth
read -DHAVE_CONFIG_H -DLINUX=2 -D_REENTRANT -D_GNU_SOURCE -D_LARGEFILE64_SOURCE -I./includ
e -I/usr/local/src/httpd-2.2.23/srclib/apr/include/arch/unix -I./include/arch/unix -I/usr/loca
```

图8-4 编译并安装Apache

安装完成后，使用以下命令启动Apache服务，并查看一下80端口，确认80端口已经被http的进程占用。

```
[root@localhost ~]# /usr/local/apache/bin/apachectl start
[root@localhost ~]# lsof -i:80
```


COMMAND	PID	USER	FD	TYPE	DEVICE	SIZE	NODE	NAME
httpd	7149	root	3u	IPv6	59986		TCP	*:http (LIST
httpd	7150	daemon	3u	IPv6	59986		TCP	*:http (LIST
httpd	7151	daemon	3u	IPv6	59986		TCP	*:http (LIST
httpd	7152	daemon	3u	IPv6	59986		TCP	*:http (LIST
httpd	7153	daemon	3u	IPv6	59986		TCP	*:http (LIST
httpd	7154	daemon	3u	IPv6	59986		TCP	*:http (LIST

最后，使用浏览器访问一下服务器的IP（使用ifconfig命令查看服务器IP），如果你看到如图8-5所示的界面，说明安装成功了。

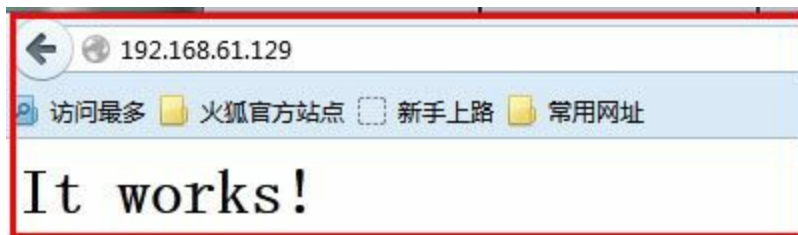


图8-5 访问Apache

8.2 RPM安装软件

前面编译安装Apache的演示看似简单，但是实际上只是因为使用了比较简单的编译方式，其实源码编译安装软件是存在不少弊端的，对于初学者而言也是一种挑战。首先，源码编译的前提是系统中安装了gcc工具，对于注重安全生产环境的用户而言是不应该安装这个工具的；其次，源码编译本身有很多可选参数，这些参数就类似于不同的开关，需要什么功能就打开相应的开关，所以需求不同往往编译参数也会相差很大，如果在编译的时候忘记了什么参数，最坏的结果就是需要重新再做编译安装（有些功能是通过添加模块启用的，这类可以不重新编译整个项目，只需要编译相应的模块即可）；再次，由于编译安装过程耗时较长，所以不适于大规模部署（有些软件单次编译需要耗时几十分钟甚至更久）；最后，源码编译无法完成软件管理功能（安装、卸载、升级等）。为了解决上述问题，RedHat采用了RPM包管理机制，并广泛用于Fedora、Mandriva、Suse等其他著名Linux发行版中。

8.2.1 什么是RPM

RPM是RedHat Package Manager的简写，顾名思义是红帽软件包管理器的意思。RPM通过一套本地数据库提供了一种更简单的软件安装管理方式，从而使得不管是安装、升级还是卸载都较源码包安装更智能。比如说在初次安装某软件的时候会提醒我们需要预先安装其他什么软件，升级的时候也会智能地保存原先的配置文件，而在卸载的时候则能视情况保留重要的数据文件等。由于Linux中一切皆文件，所以说白了，RMP其实是一种集成了文件管理和软件版本控制的工具。

RPM分为两类，第一类是二进制安装包（也就是预编译包）。事实上，如果将编译好的软件复制到相同软件环境（内核版本一致、软硬件运行环境一致）的服务器中，只要软件在原编译机中能运行，那么在新主机中也同样可以运行。而RPM采用的就是类似的方式，在特定的kernel版本下预先编译好软件（编译时使用了大多数常见的编译参数），并将所需要的文件（二进制程序、模块、配置文件等）整体打包，在新主机中安装该RPM包时，再将文件解压并复制到特定的目录中去。第二类是RPM源码包，当希望自定义编译参数，自行制作二进制安装包的时候使用。

8.2.2 RPM包管理命令：rpm

使用RPM包管理的方式是通过rpm命令。该命令常见的参数如下：

```
安装参数
-i, --install
安装软件
-v, --verbose
打印详细信息
-h, --hash
使用"#
"号打印安装进度（需要和-v
同时用）
-e, --erase
删除软件
-U, --upgrade=<packagefile>+
升级软件
--replacepkge
如果软件已经安装，则强行安装
--test
安装测试，并不实际安装
--nodeps
忽略软件包的依赖关系强行安装
--force
忽略软件包及文件的冲突
查询参数(
需要使用-q
或--query
参数)
-a, --all
查询所有安装软件
-p, --package
查询某个安装软件
-l, --list
列出某个软件包所包含的所有文件
-f, --file
查询某个文件的所属包
```

上面列出的参数比较琐碎，在实际使用中，往往需要组合使用。下面列出了rpm命令常见参数的使用方法，其中

PACKAGE_NAME代表某个包的名字，VERSION代表版本。

1) 安装软件包。

```
[root@localhost ~]# rpm -ivh PACKAGE_NAME-VERSION.rpm
```

2) 测试安装软件包，不做真实的安装。

```
[root@localhost ~]# rpm -ivh --test PACKAGE_NAME-VERSION.rpm
```

3) 安装软件包，并重新定义安装路径。

```
[root@localhost ~]# rpm -ivh --relocate /=usr/local/PACKAGE_NAME PACKAGE_NAME-VERSION.rpm
```

4) 强行安装软件包，忽略依赖关系。

```
[root@localhost ~]# rpm -ivh PACKAGE_NAME-VERSION.rpm --force --nodeps
```

5) 升级软件包。

```
[root@localhost ~]# rpm -Uvh PACKAGE_NAME-VERSION.rpm
```

6) 强行升级软件包，忽略依赖关系。

```
[root@localhost ~]# rpm -Uvh PACKAGE_NAME-VERSION.rpm --force --nodeps
```

7) 删除软件包，并忽略依赖关系。

```
[root@localhost ~]# rpm -e PACKAGE_NAME --nodeps #
```

只是包名，不需要跟版本号

8) 导入签名。

```
[root@localhost ~]# rpm --import RPM-GPG-KEY
```

9) 查询某个包是否已经安装。

```
[root@localhost ~]# rpm -q PACKAGE_NAME
```

10) 查询系统中所有已安装的包。

```
[root@localhost ~]# rpm -qa
```

11) 查询某个文件属于哪个包。

```
[root@localhost ~]# rpm -qf /etc/auto.misc
```

12) 查询某个已安装软件所包含的所有文件。

```
[root@localhost ~]# rpm -ql PACKAGE_NAME
```

13) 查询某个包的依赖关系。

```
[root@localhost ~]# rpm -qpR PACKAGE_NAME-VERSION.rpm
```

14) 查询某个包的信息。

```
[root@localhost ~]# rpm -qpi PACKAGE_NAME-VERSION.rpm
```

15) 删除软件包。

```
[root@localhost ~]# rpm -e PACKAGE_NAME
```

8.2.3 包依赖关系

在本章前面的内容中，连续几次提到了“包依赖”这个词，但均未做过多的解释。一方面是从字面意义上可以大概猜到其含义，不难理解，另一方面是当时谈论还为时太早，不过现在可以正式向大家介绍它了。

所谓包依赖，就是说在安装A包之前需要已经安装了B包，其实质是A软件运行时需要依赖B软件提供的功能。比如说openssh这个工具用于远程连接服务器，而ssh客户端和服务端之间的通信必须是加密的，但是openssh本身没必要再实现一次加密算法，只需要借助openssl提供的加密功能就可以了，这样安装openssh之前就需要已经安装openssl。那么，在这种情况下安装包时怎样才能知道需要提前安装哪些必要的包呢？事实上，如果依赖关系不满足，RPM会自动提示，而且也会拒绝安装未满足依赖关系的包。但是，大多数时候这些提示都会比较模糊，有时候你不得不根据RPM给出的一些信息，借助于一些搜索工具来判断具体的包名，而这对于很多新手来说确实有一定难度。下一节中将会具体演示安装gcc时由于包依赖关系造成的问题。

8.2.4 使用RPM包安装gcc

在CentOS和RedHat中，想要安装gcc工具，首先需要将原先的安装光盘载入光驱中。如果读者使用的是VMware虚拟机，则需要确认当前虚拟机设置中已经载入了安装光盘（ISO文件），如图8-6所示。

确认光盘确实已经挂载后，进入光盘中放置RPM包的目录。以下目录是本书中用于演示使用rpm安装的默认目录，后面不赘述。

```
#  
如果是CentOS  
系统则进入/misc/cd/CentOS  
目录，本演示使用CentOS  
[root@localhost ~]# cd /misc/cd/CentOS  
#  
如果是RedHat  
系统则进入/misc/cd/Server  
目录  
#[root@localhost ~]# cd /misc/cd/Server
```



图8-6 给虚拟机添加光驱设备

使用以下的命令安装gcc，这里需要注意的是，如果你使用的Linux的版本号不是5.5，则有可能你看到的包的版本和此处的演示有所不同除此之外是完全一致的，读者可以输入软件的名称，并在第一个“-”字符后，使用Tab键自动补全后面的内容（即软件版本）。从图8-7所示的输出内容可以看出，由于Failed dependence的原因，安装并没有成功。RPM提示需要安装cpp、glibc-devel和libgomp三个包，也就是说，在安装glibc-devel的时候遇到了依赖glibc-headers的问题，不过libgomp和cpp并不存在依赖关系，可以正常安装。

```
[root@localhost CentOS]# rpm -ivh gcc-4.1.2-48.el5.i386.rpm
warning: gcc-4.1.2-48.el5.i386.rpm: Header V3 DSA signature: NOKEY, key ID e8562897
error: Failed dependencies:
    cpp = 4.1.2-48.el5 is needed by gcc-4.1.2-48.el5.i386
    glibc-devel >= 2.2.90-12 is needed by gcc-4.1.2-48.el5.i386
    libgomp >= 4.1.2-48.el5 is needed by gcc-4.1.2-48.el5.i386
[root@localhost CentOS]# rpm -ivh cpp-4.1.2-48.el5.i386.rpm
warning: cpp-4.1.2-48.el5.i386.rpm: Header V3 DSA signature: NOKEY, key ID e8562897
Preparing...
1:cpp
[root@localhost CentOS]# rpm -ivh glibc-devel-2.5-49.i386.rpm
warning: glibc-devel-2.5-49.i386.rpm: Header V3 DSA signature: NOKEY, key ID e8562897
error: Failed dependencies:
    glibc-headers is needed by glibc-devel-2.5-49.i386
    glibc-headers = 2.5-49 is needed by glibc-devel-2.5-49.i386
[root@localhost CentOS]# rpm -ivh libgomp-4.4.0-6.el5.i386.rpm
warning: libgomp-4.4.0-6.el5.i386.rpm: Header V3 DSA signature: NOKEY, key ID e8562897
Preparing...
1:libgomp
[root@localhost CentOS]#
```

图8-7 使用rpm安装gcc（一）

接下来安装glibc-headers时，又需要先安装kernel-headers。安装kernel-headers的时候没有遇到什么问题，如图8-8所示。

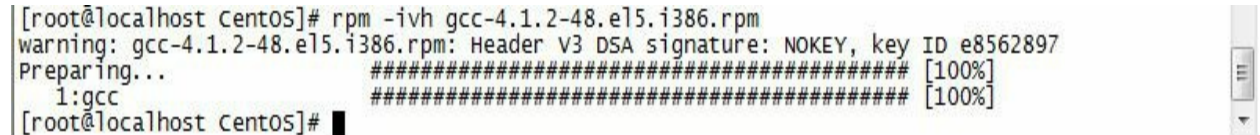
```
[root@localhost CentOS]# rpm -ivh glibc-headers-2.5-49.i386.rpm
warning: glibc-headers-2.5-49.i386.rpm: Header V3 DSA signature: NOKEY, key ID e8562897
error: Failed dependencies:
    kernel-headers is needed by glibc-headers-2.5-49.i386
    kernel-headers >= 2.2.1 is needed by glibc-headers-2.5-49.i386
[root@localhost CentOS]# rpm -ivh kernel-headers-2.6.18-194.el5.i386.rpm
warning: kernel-headers-2.6.18-194.el5.i386.rpm: Header V3 DSA signature: NOKEY, key ID e8562897
Preparing...
1:kernel-headers
[root@localhost CentOS]#
```

图8-8 使用rpm安装gcc（二）

解决了最下层的软件依赖关系后，可以继续从下往上安装。这里先后安装了kernel-headers、glibc-headers、glibc-devel，至此，已经解决了所有依赖关系，现在可以安装gcc了，如图8-9所示。

从以上的安装过程中可以看出，安装gcc的过程中遇到的依赖关系如下：

```
gcc-4.1.2-48.el5.i386.rpm
|
|----- cpp-4.1.2-48.el5.i386.rpm
|
|----- glibc-devel-2.5-49.i386.rpm
|               |----- glibc-headers-2.5-49.i386.rpm
|               |----- kernel-headers-2.6.18-
194.el5.i386.rpm
|
|----- libgomp-4.4.0-6.el5.i386.rpm
```



```
[root@localhost CentOS]# rpm -ivh gcc-4.1.2-48.el5.i386.rpm
warning: gcc-4.1.2-48.el5.i386.rpm: Header V3 DSA signature: NOKEY, key ID e8562897
Preparing... ##### [100%]
1: gcc ##### [100%]
[root@localhost CentOS]#
```

图8-9 使用rpm安装gcc（三）

在弄清依赖关系后，也可以用一条命令安装gcc，其中的“\”符号是为了将一条命令拆为多行。命令如下所示：

```
[root@localhost CentOS]# rpm -ivh gcc-4.1.2-
48.el5.i386.rpm \
cpp-4.1.2-48.el5.i386.rpm \
glibc-devel-2.5-49.i386.rpm \
glibc-headers-2.5-49.i386.rpm \
kernel-headers-2.6.18-194.el5.i386.rpm \
libgomp-4.4.0-6.el5.i386.rpm
Preparing... ##### [10
```



```

1:libgomp ##### [ 1
2:cpp ##### [ 3
3:kernel-
headers ##### [ 50%]
4:glibc-
headers ##### [ 67%]
5:glibc-
devel ##### [ 83%]
6:gcc ##### [10

```

最后要说明的是，依赖关系是和当前的系统环境紧密相关的。读者在动手安装gcc时，很有可能由于早先安装操作系统时过程的差异而造成软件环境差异，表现为安装同样的软件时其依赖关系不完全一致，换句话说，这里为了安装gcc又安装了其他5个包，而在不同的环境中可能需要安装更多或更少的包。

8.2.5 使用RPM包安装Apache

本节将会给大家演示如何使用RPM包来安装Apache，主要目的是为了和之前编译安装Apache的方法进行对比，从而让大家了解使用RPM包安装软件的简便性和快捷性。当然，这中间也可能会遇到包依赖的问题，笔者在安装的过程中就遇到了13个依赖包，安装命令如下：

```
[root@localhost CentOS]# rpm -ivh \
httpd-2.2.3-43.el5.centos.i386.rpm \
httpd-devel-2.2.3-43.el5.centos.i386.rpm \
apr-1.2.7-11.el5_3.1.i386.rpm \
apr-util-1.2.7-11.el5.i386.rpm \
apr-devel-1.2.7-11.el5_3.1.i386.rpm \
apr-util-devel-1.2.7-11.el5.i386.rpm \
postgresql-libs-8.1.18-2.el5_4.1.i386.rpm \
pkgconfig-0.21-2.el5.i386.rpm \
db4-devel-4.3.29-10.el5.i386.rpm \
expat-devel-1.95.8-8.3.el5_4.2.i386.rpm \
openldap-devel-2.3.43-12.el5.i386.rpm \
cyrus-sasl-devel-2.1.22-5.el5_4.3.i386.rpm
warning: httpd-2.2.3-
43.el5.centos.i386.rpm: Header V3 DSA signature: NOKEY, key I
Preparing... ##### [100%]
   1:pkgconfig ##### [ 100%]
   2:apr ##### [ 100%]
   3:cyrus-sasl-
devel ##### [ 25%]
   4:postgresql-
libs ##### [ 33%]
   5:apr-
devel ##### [ 42%]
   6:openldap-
devel ##### [ 50%]
   7:expat-
devel ##### [ 58%]
   8:db4-
devel ##### [ 67%]
   9:apr-
util ##### [ 75%]
  10:apr-util-
```

```
devel ##### [ 83%]
 11:httpd ##### [ 9
devel ##### [100%]
12:httpd-
```

这就完成了Apache安装！看起来很简单吧？确实比之前编译安装的方式简单多了。安装完成后就可以用以下的命令来启动httpd服务了。

```
[root@localhost CentOS]# service httpd start
Starting httpd: [ OK ] #
看到这里出现OK
说明启动成功了
```

需要注意的是，如果读者使用的服务器之前编译安装过Apache，那么这里需要先停止所有的httpd进程才可以（使用kill或killall命令杀掉进程），否则将会造成启动服务失败。

最后使用浏览器访问一下服务器的IP地址，如果你看到如图8-10的页面，则说明使用RPM安装的Apache已经可以对外提供网站服务了。

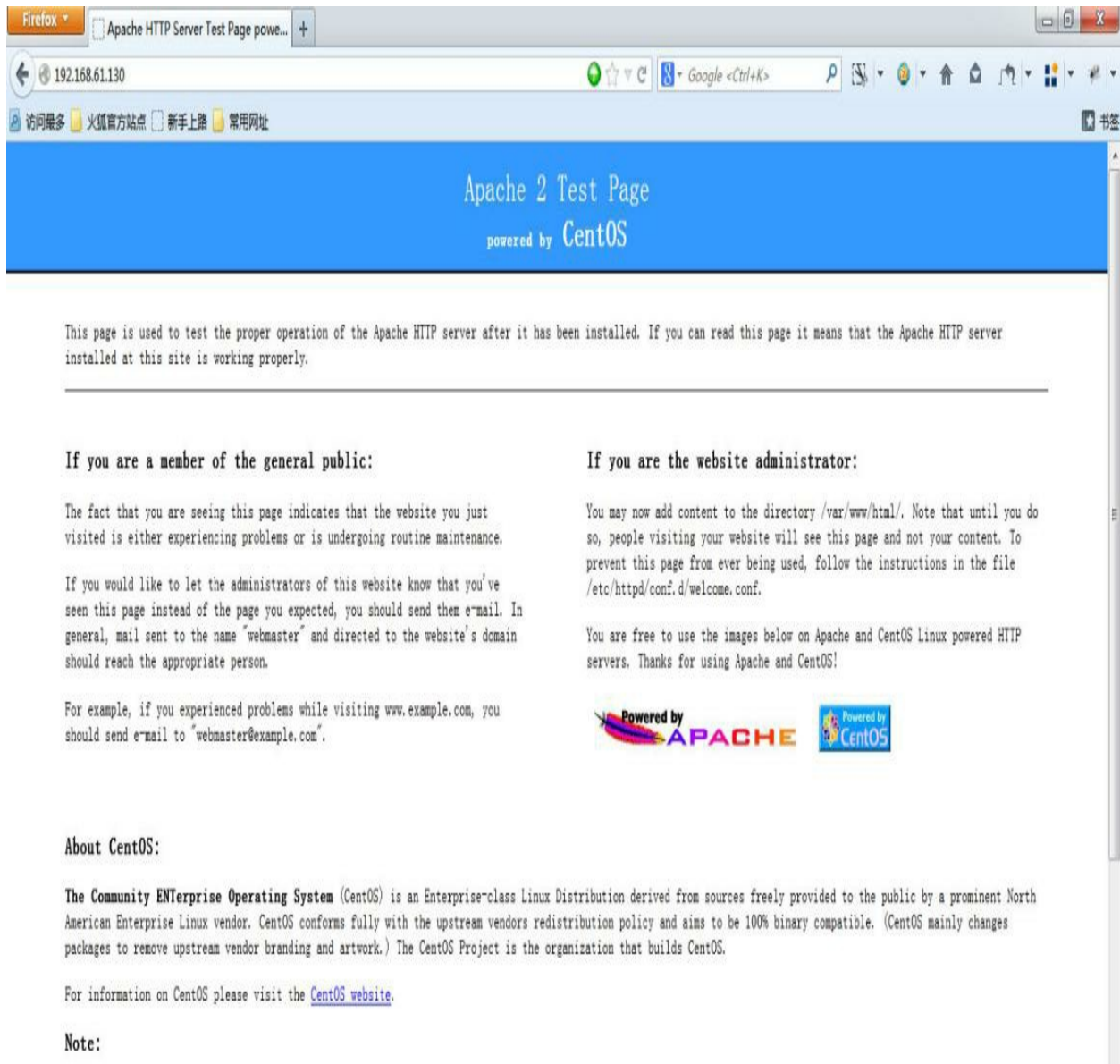


图8-10 访问Apache

8.3 yum安装软件

yum的全称为Yellow dog Updater, Modified, 是一个基于RPM的shell前端包管理器, 能够从指定的服务器上(一个或多个)自动下载并安装或更新软件、删除软件。其最大的好处是可以自动解决依赖关系。RedHat和CentOS的版本为5以上的都会默认安装yum, 所以该命令可以直接使用。

8.3.1 yum命令的基本用法

yum命令的形式一般如下。要说明的是以下演示中所使用到的PACKAGE、GROUP都是变量，需要保证运行yum命令的主机能连接外网，否则大部分命令将由于没有网络连接而不能输出结果。

```
yum [options] [command] [package]
```

```
#
```

以下演示中大写的单词是变量

1.

安装操作

```
yum install PACKAGE #
```

安装某个包

例: yum install httpd

```
yum groupinstall GROUP #
```

安装某个软件组

```
例: yum groupinstall "KDE" #
```

安装KDE

桌面

2.

升级操作

```
yum update #
```

更新系统中所有需要更新的包

```
yum update PACKAGE #
```

更新某个包

例: yum update httpd

```
yum groupupdate GROUP #
```

更新某个软件组

```
例: yum groupupdate "KDE" #
```

升级KDE

桌面

```
yum check-update #
```

检查当前系统中需要更新的包

3.

查找操作

```
yum list #
```

显示软件源中所有可用的包，一般不用

```
yum list installed #
```

显示系统中已经安装过的包

```
yum info PACKAGE #
```

显示某个包的信息

```
例: yum info httpd
yum groupinfo GROUP #
显示某个软件组的信息
例: yum groupinfo "KDE" #
显示KDE
桌面软件的信息
yum grouplist #
显示软件源宏所有的可用软件组
4.
删除操作
yum remove PACKAGE #
删除某个包
例: yum remove httpd #
删除httpd
包
yum groupremove GROUP #
删除某个软件组
例: yum groupremove "KDE" #
删除KDE
桌面
5.
清除操作
yum clean #
清除使用yum
所生成的缓存文件
```

其中options是可选参数，包括帮助参数-h，确认参数-y，静默安装参数-q等；command参数为需要进行的操作；package参数为具体的包或者软件组，按照功能分类，yum支持安装、升级、查找、删除、清理缓存等操作。

8.3.2 使用yum安装Apache

首先，本节中的演示只能在CentOS环境中进行，因为yum默认在RedHat下是无法使用的（下一节将会具体讲述这个问题的解决方法）。如果读者按照之前8.2.5节中介绍的方法，使用RPM安装了Apache，那么在使用yum安装Apache之前需要先删除前面安装的包。由于前面的安装中，有13个包存在非常复杂的包依赖关系，如果使用RPM卸载包将会非常困难（大家可以试着用rpm卸载之前安装的包，会比较麻烦），所以这里可以使用yum协助卸载。大家可以在随后的输出中看到yum同时删除了必要的依赖包。命令如下所示：

```
[root@localhost ~]# yum remove httpd apr
#
随后系统会提示你是否确定卸载，"Is this ok [y/N]"
: "
在后面输入y
#
或者使用yum remove httpd apr -y
，如果加上了后面的-y
参数则不需要在后面输入y
了
```

使用yum来安装httpd时，只需要使用命令yum install httpd即可，在开始的部分打印出的“Resolving Dependency”后面所跟的就是yum检查出的安装httpd时需要安装的依赖包，可以看出这里需要安装apr和apr-util这两个包，如图8-11所示。为什么之前用RPM进行安装时需要的依赖包比这里多呢？那是因为之前在使用RPM包安装Apache时已经安装了必要的依赖包，这里使用yum进行安装的时候已经满足相应的依赖关系了，所以只需要再安装缺失的apr包就可以了。


```

[root@localhost ~]# yum install httpd
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
* addons: mirrors.163.com
* base: mirrors.163.com
* extras: mirrors.163.com
* updates: mirrors.163.com
Setting up Install Process
Resolving Dependencies
--> Running transaction check
--> Package httpd.i386 0:2.2.3-76.el5.centos set to be updated
--> Processing Dependency: libapr-1.so.0 for package: httpd
--> Processing Dependency: libaprutil-1.so.0 for package: httpd
--> Running transaction check
--> Package apr.i386 0:1.2.7-11.el5_6.5 set to be updated
--> Package apr-util.i386 0:1.2.7-11.el5_5.2 set to be updated
--> Finished Dependency Resolution

Dependencies Resolved

=====
Package Arch Version Repository Size
=====
Installing:
httpd i386 2.2.3-76.el5.centos updates 1.2 M
Installing for dependencies:
apr i386 1.2.7-11.el5_6.5 base 124 k
apr-util i386 1.2.7-11.el5_5.2 base 80 k
=====

Transaction Summary
=====
Install 3 Package(s)
Upgrade 0 Package(s)

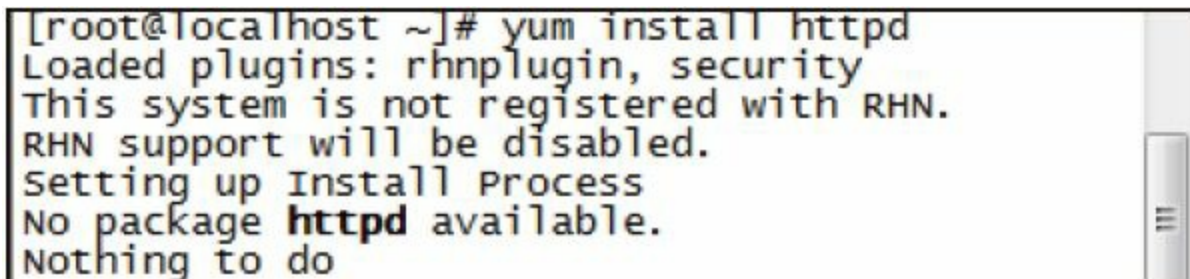
Total download size: 1.4 M
Is this ok [y/N]: y

```

图8-11 使用yum安装Apache

8.3.3 RedHat使用yum的问题

默认情况下RedHat会因为未注册RHN而无法使用yum，运行yum时将会显示如图8-12所示的信息。如果你的英语不算太差，应该能读懂。

A terminal window showing the output of the command 'yum install httpd'. The output indicates that the system is not registered with RHN, RHN support is disabled, and the package 'httpd' is not available.

```
[root@localhost ~]# yum install httpd
Loaded plugins: rhnplugin, security
This system is not registered with RHN.
RHN support will be disabled.
Setting up Install Process
No package httpd available.
Nothing to do
```

图8-12 RedHat默认无法使用yum

为了解决这个问题，只需要删除原始系统中/etc/yum.repos.d/目录下的所有repo文件，然后更换成CentOS的源即可，步骤如下：

```
#
删除系统默认的repo
文件
[root@localhost ~]# cd /etc/yum.repos.d
[root@localhost yum.repos.d]# rm rhel-debuginfo.repo
#
创建新文件CentOS.repo
，内容如下（读者只需要照抄即可）
[base]
name=CentOS-5 - Base
baseurl=http://centos.ustc.edu.cn/centos/5/os/$basearch/
gpgcheck=1
gpgkey=http://mirror.centos.org/centos/RPM-GPG-KEY-CentOS-5
#released updates
[update]
name=CentOS-5 - Updates
baseurl=http://centos.ustc.edu.cn/centos/5/updates/$basearch/
gpgcheck=1
gpgkey=http://mirror.centos.org/centos/RPM-GPG-KEY-CentOS-5
#packages used/produced in the build but not released
```

```
[addons]
name=CentOS-5 - Addons
baseurl=http://centos.ustc.edu.cn/centos/5/addons/$basearch/
gpgcheck=1
gpgkey=http://mirror.centos.org/centos/RPM-GPG-KEY-CentOS-5
#additional packages that may be useful
[extras]
name=CentOS-5 - Extras
baseurl=http://centos.ustc.edu.cn/centos/5/extras/$basearch/
gpgcheck=1
gpgkey=http://mirror.centos.org/centos/RPM-GPG-KEY-CentOS-5
#additional packages that extend functionality of existing pa
[centosplus]
name=CentOS-5 - Plus
baseurl=http://centos.ustc.edu.cn/centos/5/centosplus/$basear
gpgcheck=1
enabled=0
gpgkey=http://mirror.centos.org/centos/RPM-GPG-KEY-CentOS-5
#contrib - packages by Centos Users
[contrib]
name=CentOS-5 - Contrib
baseurl=http://centos.ustc.edu.cn/centos/5/contrib/$basearch/
gpgcheck=1
enabled=0
gpgkey=http://mirror.centos.org/centos/RPM-GPG-KEY-CentOS-5
```

设置完成后就可以使用新安装的yum来安装软件了。首先使用yum clean all&&yum makecache刷新缓存，然后输入yum install httpd看一下效果，如图8-13所示。至此我们成功地利用“偷梁换柱”的方法解决了RedHat系统中默认不能使用yum工具的问题。

```
[root@localhost yum.repos.d]# yum install httpd
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
Setting up Install Process
Resolving Dependencies
--> Running transaction check
---> Package httpd.i386 0:2.2.3-76.el5.centos set to be updated
--> Processing Dependency: libapr-1.so.0 for package: httpd
--> Processing Dependency: libaprutil-1.so.0 for package: httpd
--> Running transaction check
---> Package apr.i386 0:1.2.7-11.el5_6.5 set to be updated
---> Package apr-util.i386 0:1.2.7-11.el5_5.2 set to be updated
--> Processing Dependency: libpq.so.4 for package: apr-util
--> Running transaction check
---> Package postgresql-libs.i386 0:8.1.23-6.el5_8 set to be updated
--> Finished Dependency Resolution

Dependencies Resolved

=====
Package Arch Version Repository Size
=====
Installing:
httpd i386 2.2.3-76.el5.centos update 1.2 M
Installing for dependencies:
apr i386 1.2.7-11.el5_6.5 base 124 k
apr-util i386 1.2.7-11.el5_5.2 base 80 k
postgresql-libs i386 8.1.23-6.el5_8 base 197 k
=====
Transaction Summary
=====
Install 4 Package(s)
Upgrade 0 Package(s)

Total download size: 1.6 M
Is this ok [y/N]: y
```

图8-13 RedHat使用yum安装Apache

有时候通过以上改变软件源的方式也仍然无法使用yum工具，那么此时将会麻烦一点。在这种情况下，改完软件源后还需要将RedHat系统上的yum工具更换成CentOS的版本才可以使用。具体步骤如下所示：

首先需要将目前系统中的yum工具删除掉。由于其依赖关系比较复杂，所以要删除的软件包比较多。笔者使用的是32位的RedHat5.5系统，需要删除的软件一共有如下11个：

```
[root@localhost ~]# rpm -e yum-3.2.22-26.el5 \
yum-metadata-parser-1.1.2-3.el5 \
yum-updatesd-0.9-2.el5 \
```

```
yum-security-1.1.16-13.el5 \  
yum-rhn-plugin-0.5.4-15.el5 \  
rhn-client-tools-0.4.20-33.el5 \  
rhn-check-0.4.20-33.el5 \  
rhn-setup-0.4.20-33.el5 \  
rhn-setup-gnome \  
pirut \  
rhnsd
```

删除完成后，直接从公网上安装CentOS的yum工具（rpm命令后可以直接跟RPM包的URL地址，yum会自动完成这些包的下载并安装），安装完成后就可以使用yum了，如下所示：

```
[root@localhost ~]# rpm -ivh \  
http://mirrors.163.com/centos/5/os/i386/CentOS/yum-3.2.22-40.el5.centos.noarch.rpm \  
http://mirrors.163.com/centos/5/os/i386/CentOS/yum-fastestmirror-1.1.16-21.el5.centos.noarch.rpm \  
http://mirrors.163.com/centos/5/os/i386/CentOS/yum-metadata-parser-1.1.2-4.el5.i386.rpm
```

或许大家还沉浸在使用yum的神奇体验之中，当这种兴奋劲过去后随之而来的可能会是一个疑问：为什么它会这么神奇？下面两小节不仅能让你知道为什么，还能让你动手做属于自己的yum源。

8.3.4 自建本地yum源

上一节的最后我们第一次接触到了一个叫repo的文件（/etc/yum.repos.d/CentOS.repo），仔细观察这个文件不难发现，其实该文件中包含了诸多以http://开头的URL地址。事实上，这些都是可以使用浏览器访问的地址，其中\$basearch是个变量，yum会根据本地服务器的操作系统类型自行判断是i386还是x86_64。大家可以试着访问一下<http://centos.ustc.edu.cn/centos/5>，然后逐个目录查看一下。

在repo文件中，每个以方括弧开始的部分都是一个“源”，所以前面的repo文件中其实定义了base、updates、addons、extras、centosplus、contrib六个源。这里以第一部分为例进行解释，如下所示：

```
[base]
#
命名一个叫"base"
的源
name=CentOS-5 - Base
#
该源的名字叫作CentOS-5 - Base
baseurl=http://centos.ustc.edu.cn/centos/5/os/$basearch/
#
该源的http
地址，$basearch
是一个变量，其值和命令uname -m
输出一致
#baseurl
支持http
、file
、ftp
三种类型
gpgcheck=1
#
开启gpg
验证
gpgkey=http://mirror.centos.org/centos/RPM-GPG-KEY-CentOS-5
```

```
#  
定义gpgkey  
地址
```

实际上，只要是在/etc/yum.repos.d/目录中以.repo结尾的文件，都是yum认可的repo文件，所以之前的CentOS.repo文件最多可以分拆成6个独立的repo文件。是选择使用一个repo文件包含所有的源，还是每个源都独立使用一个repo文件就全看个人的喜好了，没有好坏之分。

为了建立本地源，首先需要将安装系统的光盘载入光驱中，如果使用的是虚拟机则需要保证光驱设备已经载入了相应的ISO镜像。如果是CentOS系统，默认在/etc/yum.repos.d/目录中会有CentOS-Base.repo和CentOS-Media.repo两个文件，这两个文件是CentOS的默认源，其中CentOS-Base.repo是网络源，会影响本地源，所以此时需要将CentOS-Base.repo禁用。禁用的方式很简单，只需要将该文件重命名成不以.repo结尾的文件即可（比如说CentOS-Base.repo.backup）。然后通过修改CentOS-Media.repo建立本地源，修改方式如下所示：

```
[root@localhost yum.repos.d]# cat CentOS-Media.repo  
# CentOS-Media.repo  
#  
# This repo is used to mount the default locations for a CDROM  
#                                     CentOS-  
5. You can use this repo and yum to install items directly o  
# DVD ISO that we release.  
#  
# To use this repo, put in your DVD and use it with the other  
# yum --enablerepo=c5-media [command]  
#  
# or for ONLY the media repo, do this:  
#  
# yum --disablerepo=* --enablerepo=c5-media [command]  
[c5-media]  
name=CentOS-$releasever - Media  
baseurl=file:///misc/cd/ #  
修改此行的目录
```



```
file:///media/cdrom/ #
删除此行
file:///media/cdrecorder/ #
删除此行
gpgcheck=1
enabled=1 #
这里的0
改为1
，表示启用
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-5
```

完成后，使用`yum clean all`与`yum makecache`刷新缓存，然后yum开始下载本地的repo并创建缓存，直到出现Metadata Cache Created，如图8-14所示，表示操作完成了。尝试使用本地源来安装httpd时，其输出结果如图8-15所示。

```
[root@localhost yum.repos.d]# yum clean && yum makecache
Loaded plugins: fastestmirror
Error: clean requires an option: headers, packages, metadata, dbcache, plugins, expire-cache, all
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
c5-media | 1.1 kB 00:00
Metadata Cache Created
[root@localhost yum.repos.d]# yum clean all
Loaded plugins: fastestmirror
Cleaning up Everything
Cleaning up list of fastest mirrors
[root@localhost yum.repos.d]# yum makecache
Loaded plugins: fastestmirror
Determining fastest mirrors
c5-media | 1.1 kB 00:00
c5-media/filelists | 2.9 MB 00:00
c5-media/other | 9.1 MB 00:00
c5-media/group | 920 kB 00:00
c5-media/primary | 920 kB 00:00
c5-media 2599/2599
c5-media 2599/2599
c5-media 2599/2599
Metadata Cache Created
[root@localhost yum.repos.d]#
```

图8-14 CentOS重建yum缓存


```
[root@localhost yum.repos.d]# yum install httpd
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
Setting up Install Process
Resolving Dependencies
--> Running transaction check
---> Package httpd.i386 0:2.2.3-43.el5.centos set to be updated
--> Processing Dependency: libapr-1.so.0 for package: httpd
--> Processing Dependency: libaprutil-1.so.0 for package: httpd
--> Running transaction check
---> Package apr.i386 0:1.2.7-11.el5_3.1 set to be updated
---> Package apr-util.i386 0:1.2.7-11.el5 set to be updated
--> Finished Dependency Resolution

Dependencies Resolved

=====
Package                Arch             Version           Repository        Size
=====
Installing:
httpd                   i386             2.2.3-43.el5.centos  c5-media          1.2 M
Installing for dependencies:
apr                     i386             1.2.7-11.el5_3.1   c5-media          123 k
apr-util                i386             1.2.7-11.el5       c5-media           80 k
=====
Transaction Summary
=====
Install      3 Package(s)
Upgrade      0 Package(s)

Total download size: 1.4 M
Is this ok [y/N]: y
```

图8-15 使用本地yum安装httpd

如果是在RedHat系统中制作本地源，那么步骤就会略有不同。RedHat安装介质的根目录中并没有repodata目录（这个目录是yum在baseurl中的根目录中可找到，里面有很多格式化的文件），而是在Cluster、ClusterStorage、Server、VT这4个目录中分别放置repodata目录，所以repo文件会有很大不同。首先，创建文件/etc/yum.repos.d/RedHat-Media.repo，内容如下：

```
[root@localhost yum.repos.d]# cat RedHat-Media.repo
[Cluster]
name=RedHat Cluster
baseurl=file:///misc/cd/Cluster
enabled=1
gpgcheck=0
[ClusterStorage]
name=RedHat ClusterStorage
baseurl=file:///misc/cd/ClusterStorage
```

```
enabled=1
gpgcheck=0
[Server]
name=RedHat Server
baseurl=file:///misc/cd/Server
enabled=1
gpgcheck=0
[VT]
name=RedHat VT
baseurl=file:///misc/cd/VT
enabled=1
gpgcheck=0
```

完成后，使用`yum clean all`与`yum makecache`刷新缓存，这时`yum`命令就可以使用本地源安装软件了（为了不影响实验效果，注意将`/etc/yum.repos.d/`目录中的其他`repo`文件移出目录，或修改为不以`.repo`结尾的文件名），如图8-16所示。

```

[root@localhost yum.repos.d]# yum clean all && yum makecache
Loaded plugins: rhnplugin, security
Cleaning up Everything
Loaded plugins: rhnplugin, security
This system is not registered with RHN.
RHN support will be disabled.
Cluster
Cluster/filelists
Cluster/other
Cluster/group
Cluster/primary
ClusterStorage
ClusterStorage/filelists
ClusterStorage/other
ClusterStorage/group
ClusterStorage/primary
Server
Server/filelists
Server/other
Server/group
Server/primary
VT
VT/filelists
VT/other
VT/group
VT/primary
Cluster
ClusterStorage
Server
VT
Cluster
ClusterStorage
Server
VT
Cluster
ClusterStorage
Server
VT
Metadata Cache Created
[root@localhost yum.repos.d]#

```

1.3 kB	00:00
110 kB	00:00
25 kB	00:00
101 kB	00:00
6.5 kB	00:00
1.3 kB	00:00
11 kB	00:00
12 kB	00:00
105 kB	00:00
9.0 kB	00:00
1.3 kB	00:00
2.4 MB	00:00
6.7 MB	00:00
1.0 MB	00:00
753 kB	00:00
1.3 kB	00:00
26 kB	00:00
47 kB	00:00
100 kB	00:00
9.0 kB	00:00
	32/32
	39/39
	2348/2348
	36/36
	32/32
	39/39
	2348/2348
	36/36
	32/32
	39/39
	2348/2348
	36/36

图8-16 RedHat重建本地缓存

8.3.5 自建网络yum源

如果你在公司只维护一到两台Linux服务器，那么使用本地yum源或许是可以接受的。但是如果运维大规模的服务器，使用本地源就不现实了。作为一个Linux工程师，不能把技术活做成体力活。在这种情况下，最简单的方式自然是用官方源——这样连本地源都不需要了，但也总是会有各种各样的原因不能这么做：首先，你无法控制官方源的版本更新，由于大量的机器都是用于线上生产的，如果由于某种原因更新了某个软件包，很可能造成系统无法正常工作，而使用自己创建的源则可以严格控制软件版本；其次，很多公司有自己定制开发的软件包，这些软件包只能通过内部的yum源服务器提供安装和更新；最后，对于很多生产服务器来说是没有外网权限的，只能使用内部yum源服务。鉴于以上原因，知道如何自建网络yum源是非常有必要的。本小节将演示使用CentOS系统做源服务器和使用CentOS、RedHat系统互做源服务器的场景，用于演示的两种系统的发行版版本（5.5）和操作系统类型（i386）都是一致的（读者考虑一下为什么）。

不管是使用CentOS还是RedHat做源服务器，所需的步骤如下：

- 1) 安装Apache服务（提供http协议的共享源）；
- 2) 将安装介质中的内容共享出来；
- 3) 在客户机上配置对应的repo文件（repo文件的内容需要根据源的内容做相应的调整）。

下面首先演示使用CentOS作为源服务器的场景（该服务器的IP为192.168.61.130）。首先安装Apache，该步骤请读者自行完成（使用RPM或者yum安装，安装完成后启动httpd服

务)。安装完成后，apache的文档目录默认是/var/www/html，为了能访问光盘安装介质中的文件，可以有两种方式，一种方式是把/misc/cd目录中的所有文件复制到/var/www/html中；还有更为简单一种方式，即做软链接，如下所示：

```
[root@localhost ~]# cd /var/www/html
[root@localhost html]# ln -s /misc/cd/ .
[root@localhost html]# ls -l #
看到软链接已经做好了
total 0
lrwxrwxrwx 1 root root 9 Feb 25 21:06 cd -> /misc/cd/
```

使用浏览器访问该服务器的http://IP/cd来测试apache是否成功共享安装文件，如果一切正常，应该会看到如图8-17所示的界面。

至此源服务器就设置好了。接下来使用一台服务器作为客户端来测试一下该源服务器是否可以使用（客户端服务器为RedHat系统，IP为192.168.61.131），操作系统可以是RedHat或CentOS，只需要保证操作系统是32位、版本为5.5（因为现在的源只能给这种版本的操作系统使用）即可。按照如下方式创建FirstYum.repo，然后更新一下yum缓存，如果成功就可以看到如图8-18所示的界面，也能够成功安装软件。注意，软件源是来自FirstYum，如图8-19所示。

```
[root@localhost yum.repos.d]# cat FirstYum.repo
[FirstYum]
name=CentOS-5 - FirstYum
baseurl=http://192.168.61.130/cd
gpgcheck=1
gpgkey=http://192.168.61.130/cd/RPM-GPG-KEY-CentOS-5
```

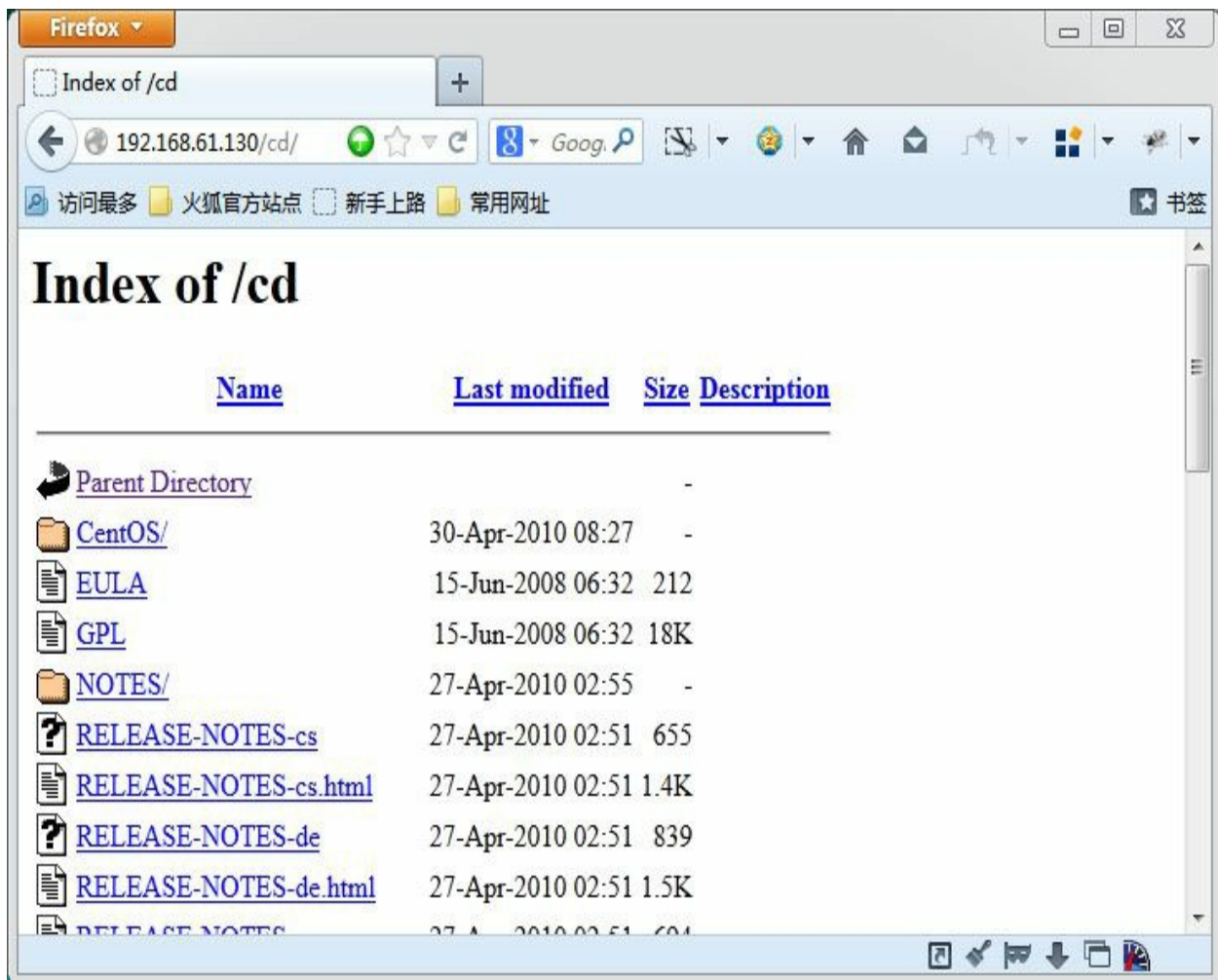


图8-17 查看共享文件

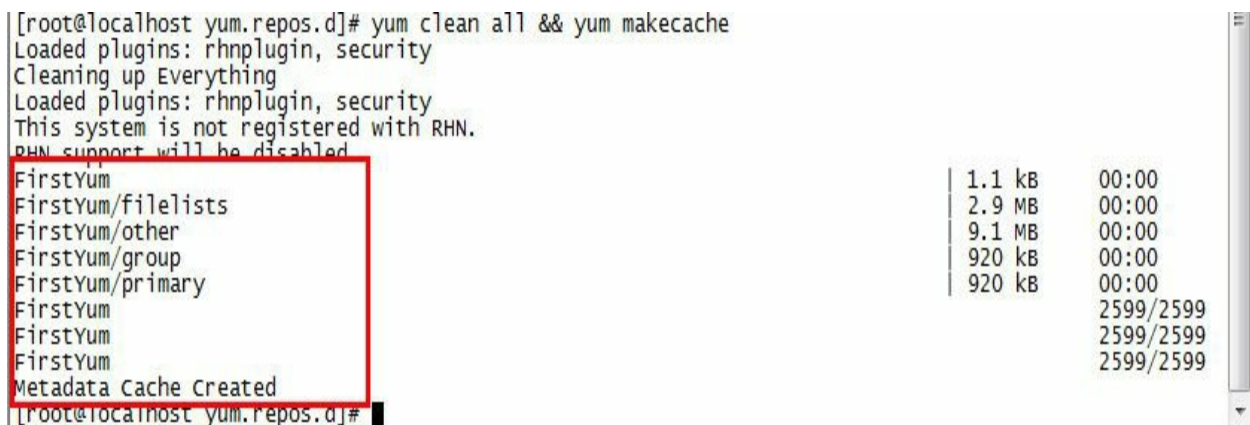


图8-18 重建yum缓存


```

[root@localhost ~]# yum install httpd
Loaded plugins: rhnplugin, security
This system is not registered with RHN.
RHN support will be disabled.
Setting up Install Process
Resolving Dependencies
--> Running transaction check
--> Package httpd.i386 0:2.2.3-43.el5.centos set to be updated
--> Processing Dependency: libapr-1.so.0 for package: httpd
--> Processing Dependency: libaprutil-1.so.0 for package: httpd
--> Running transaction check
--> Package apr.i386 0:1.2.7-11.el5_3.1 set to be updated
--> Package apr-util.i386 0:1.2.7-11.el5 set to be updated
--> Processing Dependency: libpq.so.4 for package: apr-util
--> Running transaction check
--> Package postgresql-libs.i386 0:8.1.18-2.el5_4.1 set to be updated
--> Finished Dependency Resolution

Dependencies Resolved

=====
Package Arch Version Repository Size
=====
Installing:
httpd i386 2.2.3-43.el5.centos FirstYum 1.2 M
Installing for dependencies:
apr i386 1.2.7-11.el5_3.1 FirstYum 123 k
apr-util i386 1.2.7-11.el5 FirstYum 80 k
postgresql-libs i386 8.1.18-2.el5_4.1 FirstYum 196 k
=====
Transaction Summary
=====
Install 4 Package(s)
Upgrade 0 Package(s)

Total download size: 1.6 M
Is this ok [y/N]:

```

图8-19 软件源是FirstYum

如果使用RedHat作为源服务器，服务端的配置和前面介绍的CentOS是完全一致的，只是客户端服务器上的repo文件需要修改成以下内容：

```

[root@localhost yum.repos.d]# cat SecondYum.repo
[Cluster]
name=RedHat Cluster
baseurl=http://192.168.61.131/cd/Cluster
enabled=1
gpgcheck=0
[ClusterStorage]
name=RedHat ClusterStorage
baseurl=http://192.168.61.131/cd/ClusterStorage
enabled=1

```

```
gpgcheck=0
[Server]
name=RedHat Server
baseurl=http://192.168.61.131/cd/Server
enabled=1
gpgcheck=0
[VT]
name=RedHat VT
baseurl=http://192.168.61.131/cd/VT
enabled=1
gpgcheck=0
```

如果读者在网上搜索自建网络yum源的相关文档，可能会发现其中十有八九都有使用createrepo工具重新创建repodata这一步骤，其实这一步不是必要的。事实上，repodata是当前所有RPM包依赖关系的索引，只有在RPM包的目录中放置的文件有经过修改（添加、删除或修改了其中的RPM包）时，才需要重建repodata。这里并不涉及任何包的修改，所以即便是重新创建repodata，其内容和之前的repodata也是一致的。而且，在本例中，由于光驱是只读文件系统，光盘中的所有文件都无法修改，而且也不能重建（重建repo需要写repodata目录）。

8.4 三种安装方法的比较

至此，我们已经学习了源码编译安装、RPM安装、yum安装三种软件安装方式，从易用性、效率角度来看，这三种方式明显是呈递增的趋势，实际上这也是Linux下包管理的历史发展过程。

编译安装的好处是可以根据具体的应用场景、特定的需求，甚至是个人的喜好来量身定制软件的功能模块，而使用预编译（RPM包就是预编译的软件，所以RPM管理和yum管理都只是对这些预编译的包进行管理）的方式相对来说会显得臃肿。而且由于编译过程中，编译器会根据服务器硬件和软件环境来自动做一些优化处理，因此，相对预编译软件来说，后期在软件运行时编译安装的方式更能提升部分系统使用效率（根据不同的软件，提升率各有不同）。但是其缺点也是显而易见的，首先编译安装耗时久，不适合大量部署；其次在生产服务器上编译软件本身也是极不安全的做法，必须杜绝。

从大规模运维的角度来说，安全性、高效、易管理是排在第一位的，所以必须采取更方便的包管理方式。如果想要同时享有编译软件和包管理器的优点（高效运行，集中管理），那就必须自己预编译RPM包，同时使用包管理工具将这些包安装在同平台的服务器中，这就是下一小节中将要讲述的内容：重建RPM包。

8.5 重建RPM包

前文中提到了RPM包有两种，一种是二进制安装包，还有一种是源码包，这种包的后缀名一般以.src.rpm结束（有时简称为srpm），标识着这是一个“包含源码的RPM包”。除了上一小节最后提到的原因以外，还可能由于其他原因需要使用srpm包来重建RPM包，比如，想比Linux官方更早地修复某个软件的bug而需要自行修正软件代码，或只有源码包的软件需要作成RPM包等。但是至于是否一定要重建RPM包，也需要慎重考虑，因为一旦你这么做了，那就意味着后期后的任何更新、bug修复，都需要再次重新制作新的RPM包，这会加大维护工作。

不管是RedHat还是CentOS，都在其官方网站上提供了全部的srpm包，如果发行版为5.5，可以到<http://ftp.redhat.com/pub/redhat/linux/enterprise/5Server/en/os/SRP>（RedHat）或<http://vault.centos.org/5.5/os/SRPMS/>（CentOS）下载到。

8.5.1 创建重建环境

首先需要确定系统中存在rpmbuild命令，如果没有这个命令则会出现如下报错：

```
[root@localhost ~]# rpmbuild
-bash: rpmbuild: command not found
```

通过yum来安装这个软件，完成后再运行rpmbuild--version命令来检查是否安装成功。

```
[root@localhost ~]# yum install rpm-build
[root@localhost ~]# rpmbuild --version
RPM version 4.4.2.3
```

安装完成后，会生成/usr/src/redhat目录（由于CentOS本身是一种类RedHat的发行版，所以也会生成这个目录），并且其中包含如下5个目录：

```
[root@localhost redhat]# ll
total 20
drwxr-xr-x 2 root root 4096 Feb 25 18:06 BUILD
drwxr-xr-x 9 root root 4096 Feb 26 07:25 RPMS
drwxr-xr-x 2 root root 4096 Feb 25 18:06 SOURCES
drwxr-xr-x 2 root root 4096 Feb 25 18:06 SPECS
drwxr-xr-x 2 root root 4096 Feb 25 18:06 SRPMS
```

最后，由于通过srpm包创建RPM包实际上也是一个编译过程，所以必须保证gcc编译器和make命令已经安装了。可以通过以下命令一并安装：

```
[root@localhost ~]# yum install gcc make
```

如果确认以上的条件都满足，则具备了重建RPM的环境。

8.5.2 快速重建RPM包

重建RPM包最快速的方法是使用如下命令，但是也可能遇到包依赖的问题，只需要按照系统给出的错误提示修正即可。

```
[root@localhost ~]# rpmbuild --rebuild /PATH/TO/SRPM
```

下面演示如何重建rsh这个工具。首先需要下载srpm包，然后使用上述命令进行重建。但是在此过程中出现了如下的错误，这时只需要安装缺少的文件即可。

```
[root@localhost ~]# wget \
http://vault.centos.org/5.5/os/SRPMS/rsh-0.17-40.el5.src.rpm
[root@localhost ~]# rpmbuild --rebuild rsh-0.17-
40.el5.src.rpm
.....(
略去内容).....
error: Failed build dependencies:
        libtermcap-devel is needed by rsh-0.17-40.i386
        pam-devel is needed by rsh-0.17-40.i386
[root@localhost ~]# yum install libtermcap-devel pam-devel
.....(
此处略去yum
输出，然后再次重建).....
[root@localhost ~]# rpmbuild --rebuild rsh-0.17-
40.el5.src.rpm
```

重建完成后，在/usr/src/redhat/RPMS/i386目录中生成编译好的RPM包（如果你的服务器的操作系统架构是i686，则是在/usr/src/redhat/RPMS/i686中）。

```
[root@localhost i386]# ll
total 152
-rw-r--r-- 1 root root 74079 Feb 26 08:16 rsh-0.17-
```

40.i386.rpm
-rw-r--r-- 1 root root 67033 Feb 26 08:16 rsh-server-0.17-
40.i386.rpm

8.5.3 以spec文件重建RPM包

另一种方式是使用spec文件重建RPM包，其中spec文件是一个重建RPM包的配置文件，描述了RPM包的相关信息。同样以rsh为例，首先需要“安装”这个包，这里的安装打了引号是因为这不是真正意义上的安装，而是将源码包以及一些补丁（patch文件）解压到/usr/src/redhat/SOURCES目录，同时将一个spec文件解压到/usr/src/redhat/SPECS目录中的过程。实际上srpm包就是由源码包、补丁和spec文件组成的。

```
[root@localhost i386]# rpm -i rsh-0.17-40.el5.src.rpm
```

图8-20中显示了“安装”srpm后，在相应目录中生成的文件。

件重建的RPM包了。注意这次生成的包因为和之前快速重建生成的包名是一致的，所以需要覆盖操作（可注意观察，文件的时间戳被更新）。

```
[root@localhost i386]# ll
total 152
-rw-r--r-- 1 root root 74079 Feb 26 08:45 rsh-0.17-40.i386.rpm
-rw-r--r-- 1 root root 67033 Feb 26 08:45 rsh-server-0.17-40.i386.rpm
```

8.5.4 spec文件简介

从作用上来说，spec文件类似于源码编译时的Makefile文件，是重建rpm包的核心文件。spec文件有一定的模板格式，一般来说分为preamble（序言）、prep（前期准备）、build（编译）、install（安装）、clean（清理）、files（文件列表）、changelog（修改日志）这几个部分。

1.preamble（序言）

基础信息部分，主要包含软件包的功能描述、版本、版权、作者、制作时间等内容，比如说我们可以用rpm命令查询到之前创建的rsh-0.17-40.i386.rpm软件包的相关基础信息。

```
[root@localhost i386]# rpm -qpi rsh-0.17-40.i386.rpm
Name           : rsh                      Relocations: (not relocat
Version        : 0.17                    Vendor: (none)
Release       : 40                       Build Date: Tue 26 Feb 20
Install Date: (not installed)             Build Host: local
Group          : Applications/Internet    Source RPM: rsh-
0.17-40.src.rpm
Size           : 130504                   License: BSD
Signature      : (none)
Summary        : Clients for remote access commands (rsh, rlogin
Description    :
The rsh package contains a set of programs which allow users
commands on remote machines, login to other machines and copy
between machines (rsh, rlogin and rcp). All three of these c
use rhosts style authentication. This package contains the c
needed for all of these services.
The rsh package should be installed to enable remote access t
machines.
```

这些信息都是可以在spec文件中定义的，在spec文件中可使用特定的“关键字”来定义，常用的关键字如下：

- Summary: 包的简介。
- Name: 包的名称。
- Version: 软件版本。
- Release: 发布序列号。
- License: 软件授权, 常见的有GPL、BSD、MIT、Distributable、Commercial、Share等。
- Group: 软件分组, 常见的软件分组如下:
 - Amusements/Games (娱乐/游戏)
 - Amusements/Graphics (娱乐/图形)
 - Applications/Archiving (应用/文档)
 - Applications/Communications (应用/通信)
 - Applications/Databases (应用/数据库)
 - Applications/Editors (应用/编辑器)
 - Applications/Emulators (应用/仿真器)
 - Applications/Engineering (应用/工程)
 - Applications/File (应用/文件)
 - Applications/Internet (应用/因特网)
 - Applications/Multimedia (应用/多媒体)

- Applications/Productivity（应用/产品）
- Applications/Publishing（应用/印刷）
- Applications/System（应用/系统）
- Applications/Text（应用/文本）
- Development/Debuggers（开发/调试器）
- Development/Languages（开发/语言）
- Development/Libraries（开发/函数库）
- Development/System（开发/系统）
- Development/Tools（开发/工具）
- Documentation（文档）
- System Environment/Base（系统环境/基础）
- System Environment/Daemons（系统环境/守护）
- System Environment/Kernel（系统环境/内核）
- System Environment/Libraries（系统环境/函数库）
- System Environment/Shells（系统环境/接口）
- User Interface/Desktops（用户界面/桌面）
- User Interface/X（用户界面/X窗口）
- User Interface/X Hardware Support（用户界面/X硬件支持）

- BuildRoot: 编译使用的目录。
- BuildPrereq: 编译前需要满足的包。
- BuildRequires: 编译时需要安装的包。
- Source: 源码包。
- Patch: 补丁文件。
- Description: 更详细的描述。
- Requires: 安装该包时的依赖包。

2.prep（前期准备）

prep是预处理部分，以%prep开头，用于正式编译前的准备工作，包括删除老的源码、解压源代码（%setup）、对源码应用补丁（%patch）等操作。

%setup部分的写法一般为：

```
%setup -n %{Name}-%{Version} #  
把源码包解压到新创建的目录中
```

通常是从/usr/src/redhat/SOURCES里的包解压到/usr/src/redhat/BUILD/%{Name}-%{Version}中。注意其中的%{Name}-%{Version}是在preamble中定义的。

%patch用于对源码包打补丁，通常补丁都会与源码包在一起，一般写法为：

```
%patch -p1 #  
使用preamble  
中定义的Patch
```

补丁，-p1
是忽略patch
的第一层目录

3.build（编译）

build是正式开始编译的部分，以%build开头，相当于源码编译时的configure（配置）、make（编译）的工作，所以这部分一般由configure以及多个build命令组成。

4.install（安装）

install用于完成实际的安装过程，以%install开头，相当于源码编译时的make install，其中也会包括一些Shell的文件操作命令，包括make、cp、install、rm、mkdir等，还能定义安装该软件时需要运行的脚本，同时还能控制该脚本运行的时间（安装包之前还是之后，移除包之前还是之后）。

5.clean（清理）

clean主要适用于安装完成后的清理工作，以%clean开头，用于删除编译过程中产生的临时文件等。一般这里只需要简单地使用rm命令即可，如下所示：

```
rm -rf $RPM_BUILD_ROOT #RPM_BUILD_ROOT
```

是preamble
中定义的BuildRoot

6.files（文件列表）

files部分用于指定实际安装的文件放置的目录和相关的权限，以%files开头。这里所指定的所有文件都将会被打包到最后生成的rpm包中，这些指定的文件分为三类，

分别是说明文件（README或是changelog文件）、配置文件、可执行文件，如果在%files中不列出具体的文件，则默认包含所有文件。

在%files中，还可以使用以下字段：

- %exclude列出不被打包到rpm中的文件。

- %defattr(-,root,root)指定文件的属性，分别是mode、owner、group，-表示默认值，对文本文件是0644，可执行文件是0755。

- %attr（permissions,user,group）覆盖指定文件的权限。

- %doc指明说明文件，rpm在安装时会将这类文件复制到/usr/share/doc/%{Name}-%{Version}中。

- %config指明文件属于配置文件，在使用rpm升级软件时，会避免用rpm打包的默认配置文件覆盖原配置文件。

7.changelog（修改日志）

changelog主要是注明该软件包的开发记录，使用%changelog开头，主要作用是让开发人员了解软件开发过程以及历经的功能补全和bug修复。

第9章 vi和vim编辑器

9.1 vi和vim编辑器简介

vi编辑器是Visual Interface的简称，是Linux系统中最基本的文本编辑器，其功能与很多图形编辑器类似，可以进行编辑、查找、删除、替换等文本操作。它工作在字符模式下，而且随着其不断地更新改进，现在它已经慢慢成为一个效率很高的文本编辑工具。

vim编辑器是vi的加强版，在简单的文本操作上与vi几乎完全一致，所以习惯使用vi的人可以完全无缝地切换使用vim编辑器。同时vim还增加了很多新功能，包括代码补全、错误跳转等，可方便编程。所以vim编辑器成为了很多程序员的开发工具，和Emacs编辑器一样被称为“开发神器”。从vim的官方网站对其的介绍来看，vim也是定位成为一款“开发工具”，而不仅仅是一款文本处理工具。

从某种意义上来说，对Linux系统的管理有很大一部分就是对文本类配置文件的管理，所以学会使用一种编辑器是十分必要的。本章将详细介绍vi和vim编辑器的用法。

9.2 vi编辑器

9.2.1 模式介绍

vi编辑器有3种模式，分别是一般模式、编辑模式、末行指令模式。当使用vi打开一个文件的时候（vi命令后跟上一个文件并按回车键时的状态），就进入了一般模式。一般模式可以与编辑模式、末行指令模式相互转换，但是编辑模式和末行指令模式之间不能直接转换，必须通过一般模式进行转换。其转换过程如图9-1所示。

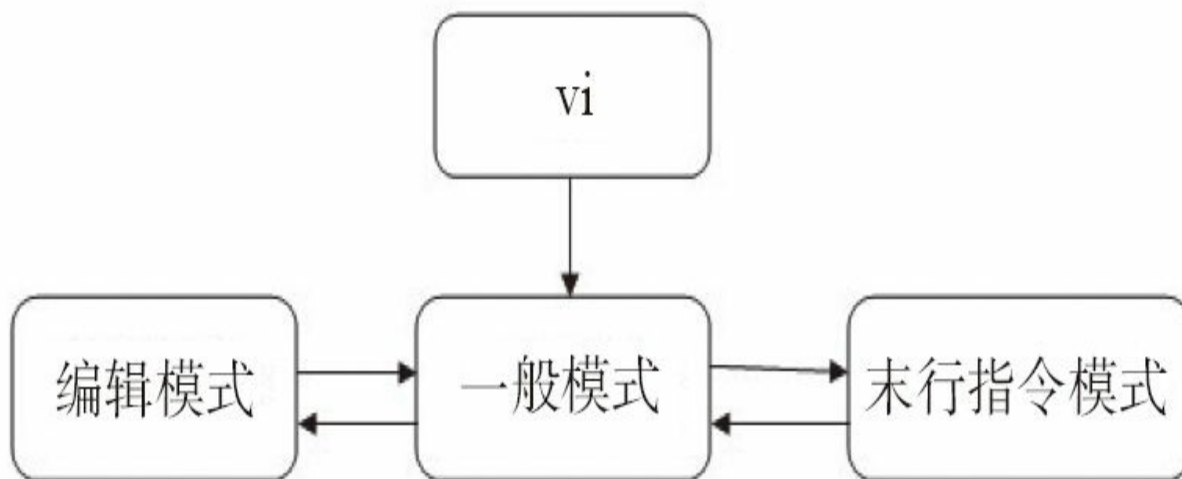


图9-1 vi编辑器的模式转换

1.一般模式

使用vi打开某个文件的时候默认进入的模式就是一般模式。在这种模式中最基础的功能就是“移动光标”——使用上下左右键来移动光标块。还可使用按键组合的方式来执行复制、粘贴、删除的功能。

2.编辑模式

在一般模式中，按i键可以进入编辑模式（这是最简单的进入方式，底部会出现“--INSERT--”字样，还有其他的进入方式后面介绍）。在编辑模式中，依然可以使用上下左右键来移动光标，同时还可以输入文字到文件中。从编辑模式回到一般模式需要按Esc键。

3.末行指令模式

在一般模式中，按冒号键（:）或斜杠键（/）或问号键（?）就会在当前视图的最后一行出现相应的符号，这就代表进入了相应的末行指令模式。

9.2.2 案例练习

案例一：使用vi创建和编辑一个文件。

1) 使用vi创建一个文件newfile，进入一般模式，如图9-2所示。

```
[root@localhost ~]# vi newfile #
```

输入该命令后按回车键便进入一般模式

2) 按i键从一般模式进入编辑模式（如图9-3所示）。



图9-2 进入vi一般模式



图9-3 进入vi编辑模式

3) 在编辑模式中写一段话后退出编辑模式，进入一般模式，如图9-4所示。

4) 在编辑模式中复制并粘贴第一行的文字，如图9-5所示。

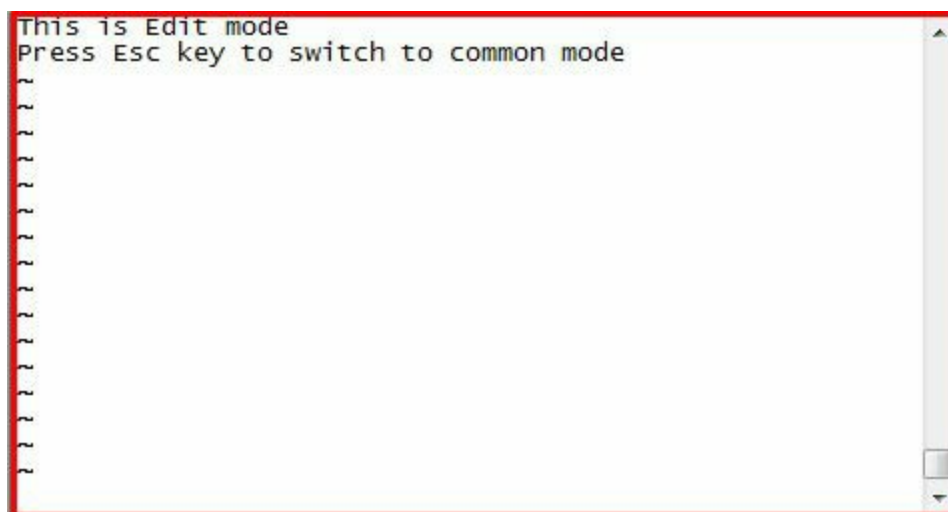


图9-4 退出vi编辑模式

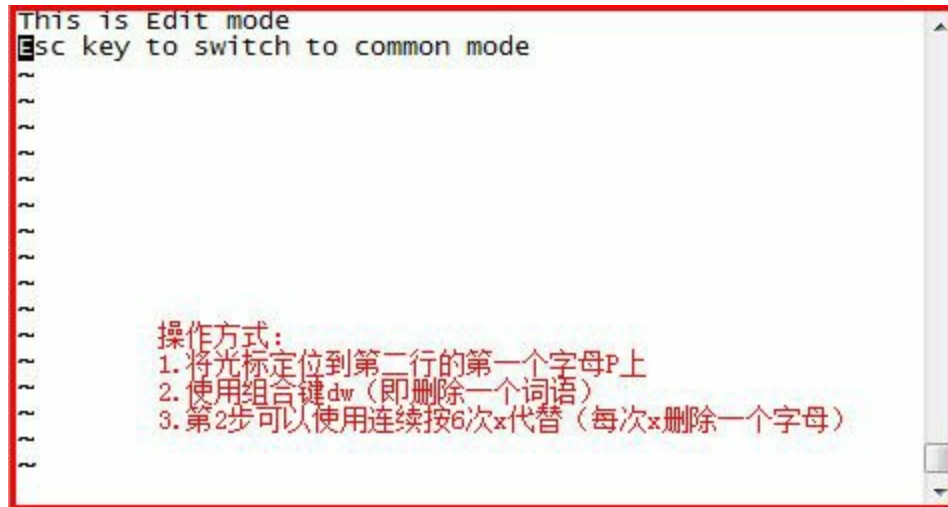


图9-7 vi删除词

7) 恢复刚刚删除的词Press，如图9-8所示。



图9-8 恢复删除的词

8) 切换至末行指令模式并保存退出，如图9-9所示。

键	动 作
h	光标左移
j	光标下移
k	光标上移
l	光标右移
\$	移动到本行的末尾
G	移动到整个文件的末尾
:n (即进入末行指令模式后输入行号回车)	移动到第 n 行
n (n 是一个数字, 按后回车)	往下移动 n 行
Ctrl+f	往下移动一页
Ctrl+b	往上移动一页
Ctrl+d	往下移动半页
Ctrl+u	往上移动半页

在之前的演示中，用到了dd组合键来删除光标所在的一行，事实上，在实现文本的删除、复制、粘贴等操作时还有其他的一些组合键，具体如表9-2所示。

表9-2 vi的编辑操作

键	动 作
n dd (n 是一个数字)	删除包含光标所在行在内的 n 行文字
dw	删除光标往后的一个单词
d\$	删除光标至最后的所有文本
x	向后删除一个字符
X	向前删除一个字符
yy	复制光标所在的行
nyy (n 是一个数字)	复制连同光标所在行在内的 n 行文字
p	将复制的文本粘贴在光标下面一行
u	撤销操作
Ctrl+r	重做操作
i	在当前光标处添加内容
I	在当前光标所在行的第一个非空处添加内容
o	在当前光标下一行插入新行并开始编辑
O	在当前光标上一行插入新行并开始编辑
a	在当前光标后一个字符开始添加内容
A	在当前光标所在行的最后一个字符处添加内容

案例二：搜索关键字。

1) 使用vi打开/etc/ssh/sshd_config文件。

```
[root@localhost ~]# vi /etc/ssh/sshd_config
```

2) 使用“/”符号查找关键字HostKey，如图9-10所示。



```
# sshd_config(5) for more information.
# This sshd was compiled with PATH=/usr/local/bin:/bin:/usr/bin
# The strategy used for options in the default sshd_config shipped with
# OpenSSH is to specify options with their default value where
# possible, but leave them commented. Uncommented options change a
# default value.
#Port 22
#Protocol 2,1
Protocol 2
#AddressFamily any
#ListenAddress 0.0.0.0
#ListenAddress ::

# HostKey for protocol version 1
#HostKey /etc/ssh/ssh_host_key
# HostKeys for protocol version 2
#HostKey /etc/ssh/ssh_host_rsa_key
#HostKey /etc/ssh/ssh_host_dsa_key

# Lifetime and size of ephemeral version 1 server key
#KeyRegenerationInterval 1h
#ServerKeyBits 768
/HostKey
```

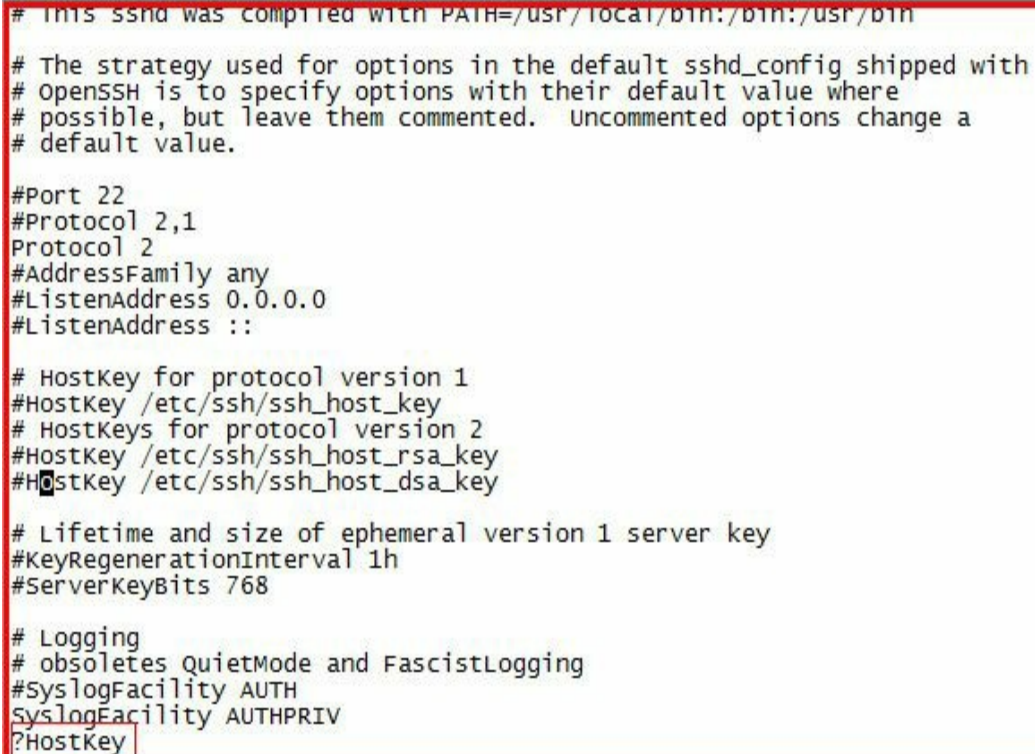
操作方式：
1. 输入/符号
2. 输入关键字HostKey

图9-10 使用/查找关键字

需要注意的是，搜索到的关键字是以当前的光标为相对位置、往下找到的第一个关键字。以图9-10为例，如果在搜索前（也就是在一般模式的时候），光标是停留在第一行的，那么搜索到的HostKey将是文本中第一次出现HostKey的地方。也就是搜索功能默认使用光标位置下移来实现搜索操作。

按照图示方法找到了第一个HostKey后，可以按n键继续往下找，每按一次光标将跳至下一个关键字处，如果要想往上寻找，则按大写字母N。

查找关键字还可以使用“?”符号，和“/”不同的是，使用“?”查找默认是从光标位置向上寻找关键字，按n键代表继续往上寻找，按N键代表向下寻找，如图9-11所示。



```
# This sshd was compiled with PATH=/usr/local/bin:/bin:/usr/bin
# The strategy used for options in the default sshd_config shipped with
# OpenSSH is to specify options with their default value where
# possible, but leave them commented. Uncommented options change a
# default value.

#Port 22
#Protocol 2,1
Protocol 2
#AddressFamily any
#ListenAddress 0.0.0.0
#ListenAddress ::

# HostKey for protocol version 1
#HostKey /etc/ssh/ssh_host_key
# HostKeys for protocol version 2
#HostKey /etc/ssh/ssh_host_rsa_key
#HostKey /etc/ssh/ssh_host_dsa_key

# Lifetime and size of ephemeral version 1 server key
#KeyRegenerationInterval 1h
#ServerKeyBits 768

# Logging
# obsoletes QuietMode and FascistLogging
#SyslogFacility AUTH
SyslogFacility AUTHPRIV
?HostKey
```

图9-11 使用?查找关键字

案例三：替换关键字。

有时候需要将整篇文档中的某个词换成另外一个词，如果靠手工寻找替换是不现实的。利用末行指令模式则可以轻易实现这个功能。为了演示这个功能，我们先做一个准备工作。

```
[root@localhost ~]# cp /etc/ssh/sshd_config /root
```

然后按照图9-12所示的方法，将/root/sshd_config文件中的HostKey全部替换成NewKey。

```
# $OpenBSD: sshd_config,v 1.73 2005/12/06 22:38:28 reyk Exp $
# This is the sshd server system-wide configuration file.  See
# sshd_config(5) for more information.
# This sshd was compiled with PATH=/usr/local/bin:/bin:/usr/bin
# The strategy used for options in the default sshd_config shipped with
# OpenSSH is to specify options with their default value where
# possible, but leave them commented.  Uncommented options change a
# default value.

#Port 22
#Protocol 2,1
Protocol 2
#AddressFamily any
#ListenAddress 0.0.0.0
#ListenAddress ::

# HostKey for protocol version 1
#HostKey /etc/ssh/ssh_host_key
# HostKeys for protocol version 2
#HostKey /etc/ssh/ssh_host_rsa_key
#HostKey /etc/ssh/ssh_host_dsa_key

:1,$s/HostKey/NewKey/g
```

图9-12 替换关键字

按回车键后，所有的HostKey就全部被替换成NewKey。替换用法的解释和其他用法如表9-3所示。

表9-3 替换用法

指 令	动 作
:n1,n2s/word1/word2/g	将 n1 到 n2 行之间的所有 word1 替换成 word2
:1,\$s/word1/word2/g	将第 1 行到最后一行的所有 word1 替换成 word2
:s/word1/word2/g	将本行的 word1 替换成 word2
:s/word1/word2	将本行第一次出现的 word1 替换成 word2

9.3 vim编辑器

9.3.1 多行编辑

既然说vim是vi的增强版，那么vim就一定有vi所没有的功能，其中之一就是vim支持多行编辑，而vi每次只能处理一行。下面还是以上节复制的/root/sshd_config文件为例，使用vim编辑该文件。

```
[root@localhost ~]# vim /root/sshd_config
```

进入一般模式后，使用Ctrl+v组合键，这时最下行会出现“--VISUAL BLOCK--”字样，这说明当前进入了Visual Block模式（如果只按大写的字母V则代表进入多行选中模式，此时最下行会出现“--VISUAL LINE--”字样）。使用上下左右键可以选中多行文字，如图9-13所示。选中后可以一次性复制（y键）、删除（d键）选中的文字或者将其粘贴到其他地方（p键）。


```
# $openBSD: sshd_config,v 1.73 2005/12/06 22:38:28 reyk Exp $
# This is the sshd server system-wide configuration file.  See
# sshd_config(5) for more information.
# This sshd was compiled with PATH=/usr/local/bin:/bin:/usr/bin
# The strategy used for options in the default sshd_config shipped with
# openSSH is to specify options with their default value where
# possible, but leave them commented.  Uncommented options change a
# default value.

#Port 22
#Protocol 2,1
Protocol 2
#AddressFamily any
#ListenAddress 0.0.0.0
#ListenAddress ::

# HostKey for protocol version 1
#HostKey /etc/ssh/ssh_host_key
# HostKeys for protocol version 2
#HostKey /etc/ssh/ssh_host_rsa_key
#HostKey /etc/ssh/ssh_host_dsa_key

-- VISUAL BLOCK --
```

11,16

Top

图9-13 vim的多行编辑

9.3.2 多文件编辑

不管是vi还是vim都可以同时打开并编辑多个文件，如同在Windows中使用Office同时打开多个文件一样。但是由于vim拥有多行编辑的功能，因此使用它在多个文件之间切换编辑的时候更加方便。本节将继续使用案例练习的方式来演示它的使用方法。准备工作如下：

```
[root@localhost ~]# touch file_a file_b
#
创建两个文件，分别是file_a
和file_b
，其内容如下
[root@localhost ~]# cat file_a
This is file_a, line 1
This is file_a, line 2
This is file_a, line 3
[root@localhost ~]# cat file_b
This is file_b, line 1
[root@localhost ~]# vim file_a file_b
#
同时打开文件file_a
和file_b
```

同时打开file_a和file_b后，默认会打开第一个文件，也就是file_a，我们把光标定位到第二行，并按V键，这时进入多行选中模式，选中第二行和第三行，并进行复制操作（按y键），如图9-14所示。



```
:files
1 # "file_a" line 2
2 %a + "file_b" line 2
Press ENTER or type command to continue
```

图9-18 vim的多文件编辑（五）

9.3.3 使用vimtutor深入学习vim

学习vim时，没有比vimtutor更好的入门教材了，输入vimtutor命令后剩下的就是跟着说明操作，整个过程不需要死记硬背，它会非常应景地告诉你应该怎么使用vim，并且全程给出了模拟演练的环境。本节总结了vimtutor提到的所有vim操作方法。

移动光标既可以用箭头键，也可以使用hjkl字母键，其中h用于左移光标，j用于下移光标，k用于上移光标，l用于右移光标。

如果使用:q!退出vim编辑器，将不保存对文本进行的修改。

如果使用:wq退出vim编辑器，将保存所有对文本进行的修改。

在一般模式下按x键删除光标所在位置的字符。

在一般模式下要在光标所在位置插入文本可输入i或a键，其中i键用于在光标前插入文本，a键用于在光标后插入文本。

在一般模式下输入dw，将从光标当前位置直到单词末尾删除，但不包括第一个字符。

在一般模式下输入de，将从光标当前位置直到单词末尾删除，但不包括最后一个字符。

在一般模式下输入d\$，将从光标当前位置直到当前行末的内容删除，且包括最后一个字符。

在一般模式下输入2w，光标将向后移动两个单词。

在一般模式下输入3e，光标将移动到后面第三个单词尾。

在一般模式下输入0（数字零），光标将移动到行首。

在一般模式下输入2dw，将删除两个单词。

在一般模式下输入dd，可以删除当前光标所在位置的一整行。

在一般模式下输入2dd，将删除当前光标位置以及下一行共计两行的内容。

在一般模式下输入u可撤销最后执行的命令，输入U可撤销对整行的修改。

在一般模式下多次输入Ctrl+R(按下Ctrl键不放开，接着按R键)，可以执行恢复命令，也就是撤销掉撤销操作。

在一般模式下按p键可将刚刚使用d操作删除的内容粘贴到当前光标所在行的下一行。

在一般模式下按r键，再输入一个字符可用新输入的字符替换光标所在位置的字符。

要从光标处改动一个单词至该单词的末尾，输入ce。

在一般模式下输入“/”符，然后输入要查找的字符串，可以在本文中查找字符串；要继续查找之前的字符串，只需要按n键；要向相反方向查找字符串，按N键即可。如果想一开始就逆向查找字符串，则用“?”代替“/”即可。

在一般模式下按“%”可以查找配对的括号)、]、或}，在程序调试时，使用这个功能用来查找不配对的括号是很有用的。

在一般模式下输入“:s/old/new/g”将会把old替换为new。要

替换两行之间出现的每个匹配串，请输入“: #, #s/old/new/g”（#, #代表的是两行的行号）。输入“: %s/old/new/g”则是替换整个文件中的每个匹配串。输入“: %s/old/new/gc”则会找出全文中的匹配内容，并询问是否替换。

在一般模式下输入“: !”然后输入一个外部命令，可以执行该外部命令。所有的外部命令都可以使用这种方式执行，命令后也可以跟必要的参数。

要将当前文件的保存到另一个文件中，请输入“: w 文件名”。

要向当前文件中插入另一个文件的内容，请输入“: r FILENAME”，其中FILENAME是另一个文件的全路径。也可以将外部命令的输出插入当前文件，例如“: r !ls”就是提取ls命令的输出并显示在当前光标处。

在一般模式下输入o键将在光标的下方插入新的一行并进入编辑模式。

输入大写R键可连续替换多个字符。注意：替换模式和编辑模式类似，只是输入的每个字符都会替换当前光标上的字符。

使用y键可复制选中的字符，用p键粘贴；可以使用yy复制整行，也可以使用yw复制一个单词。

9.4 gedit编辑器

9.4.1 gedit编辑器简介

虽然说Linux系统目前主要还是应用在服务器端，使用的编辑器主要还是非图形化的vi和vim，但是随着近几年Linux桌面化的迅猛发展，对桌面环境下的文本编辑器的需求也在逐渐增多，因此相应的工具也在发展，其中gedit就是比较著名的、老牌的图形化编辑工具之一，在很多Linux发行版中都作为系统预装的默认图形文本编辑工具。

gedit使用简单，支持包括UTF-8和GBK在内的多种字符编码（换句话说就是中文支持良好），支持远程打开文件、语法高亮、错误检查（通过安装插件），所以gedit可以作为Linux下的集成开发环境。

9.4.2 启动gedit编辑器

gedit在RedHat发行版中的打开方式是选择左上角的Applications/Accessories/Test Editor，如图9-19所示。也可通过命令启动，只需要在桌面的终端中使用命令gedit即可。gedit可以同时编辑多个文件，每个文件使用单独的标签，对于不同的程序代码也能使用多种颜色标注关键字，极大地方便了开发人员阅读和编辑代码。和Windows下很多编辑类工具相似，gedit包括File、Edit、View、Search、Tools、Documents、Help等菜单。每个菜单的含义如下。

File: 文件的创建、打开、打印、页面设置等。

Edit: 复制、粘贴、文件缓冲区操作（Redo、Undo）以及编辑器首选项配置。

View: 设置编辑文件的显示特性等。

Search: 查找、替换。

Tools: gedit的插件库。

Documents: 管理缓冲区中的文件。

Help: 帮助手册。

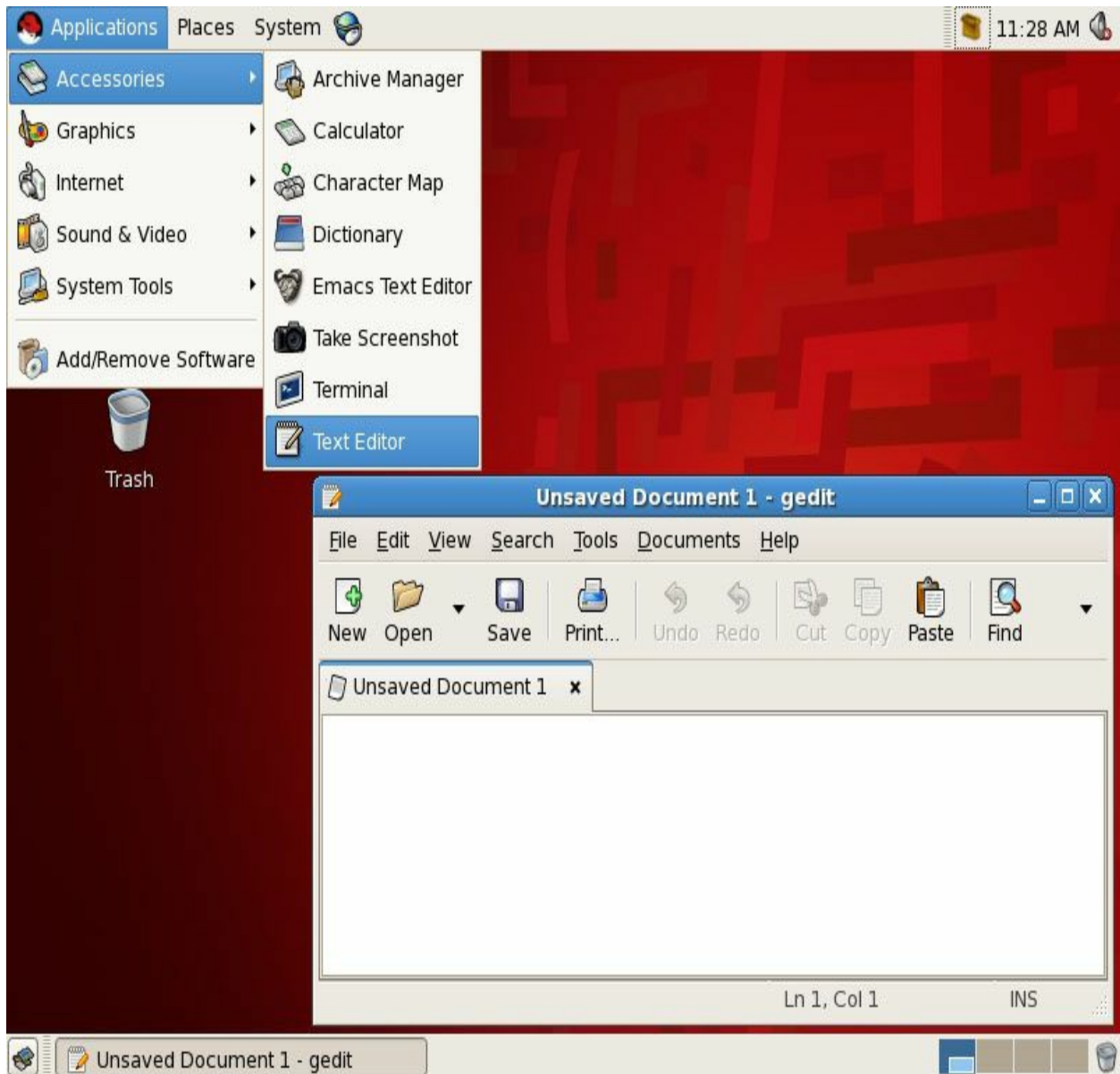


图9-19 图形界面下打开gedit的方法

第10章 正则表达式

10.1 正则表达式基础

10.1.1 什么是正则表达式

在老套机械地使用一段抽象难懂的文字来解释“什么是正则表达式”之前，让我们回忆一下自己是如何使用Office软件中的“查找”功能的。该功能似乎很简单，比如说，想要在当前文档中找到hello，只需要在查找选项中输入hello就可以了，如图10-1所示。可能大家没有意识到，其实这就是一种形式最简单的“表达式”，查找工具会使用某种匹配方式进行全文搜索，其工作的原理也非常简单，那就是先找到h，然后看后面是不是e，再看后面是不是l，以此类推。如果全部符合，那就是匹配到了。但这里可能也会出现一个问题，这种简单的查找其实也能匹配到helloworld（注意中间没有空格）这样的文字，不过Office的查找中还提供了高级功能，选中“全字匹配”就只会匹配hello了。再让我们想想数学中的方程组，它们实际上是一种表明变量关系的“表达式”，我们可以根据该表达式求出变量x、y的值，x和y可能是唯一匹配，也可能有多个匹配。

在Linux文本模式中，没有类似于Office的图形化匹配工具，但可以使用“正则表达式”来做相同的匹配工作。还是以精确匹配hello为例，在正则表达式中就可以用\<hello\>来表示，这里使用到了正则表达式的特殊符号。正则表达式中还有更多更复杂的符号可用来代表其他有意义的字符，这实际上是一种抽象的过程。提到抽象，在现实生活中我们可以用“由内燃机驱动的、有轮子的工具”来代表所有的机动车，但是计算机并不懂这些自然语言，那么用什么来代表诸如手机号、IP地址、一个网址等有一定格式和特征的字符串呢？答案就是使用正则表达式。

说到这里，再解释什么是正则表达式就显得简单明了了：正则表达式就是能用某种模式去匹配一类字符串的公式，它是由一串字符和元字符构成的字符串。所谓元字符，就是用以阐述字符表达式的内容、转换和描述各种操作信息的字符。

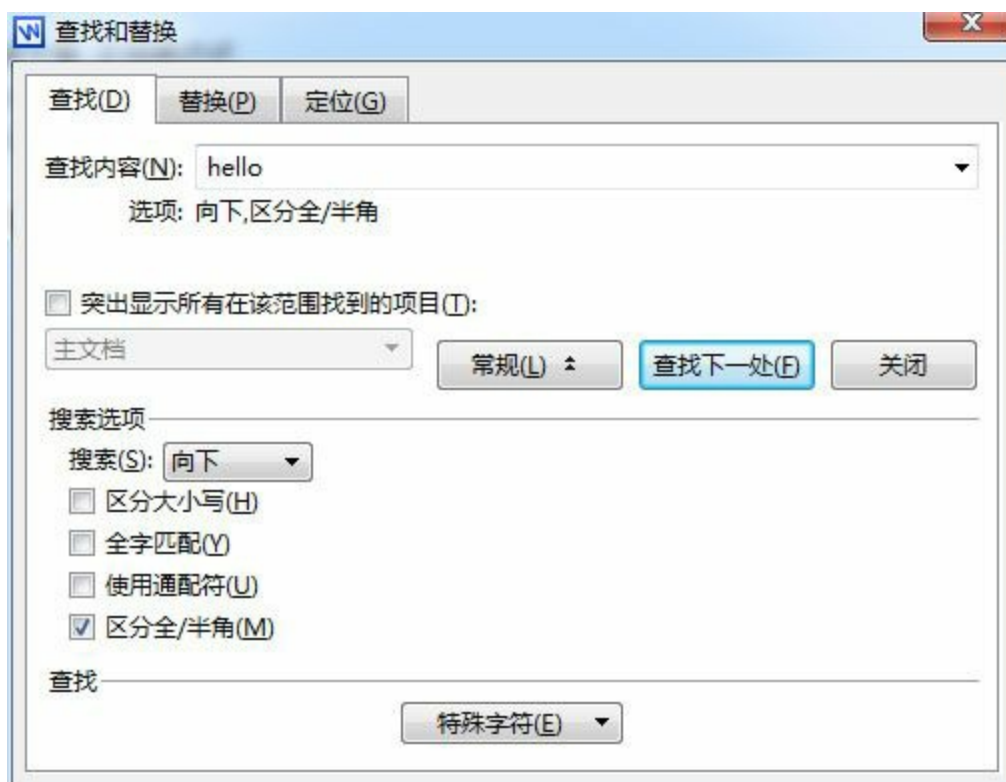


图10-1 在Office中查找文本

10.1.2 基础的正则表达式

上一节在介绍什么是正则表达式的时候，我们第一次看到了\

1.“.”（一个点）符号

点符号用于匹配除换行符之外的任意一个字符。例如：r.t可以匹配rot、rut，但是不能匹配root，若使用r..t就可以匹配root、ruut、rt（中间是两个空格）等。下面的例子是从/etc/passwd中搜索出“包含r，紧跟着两个字符，后面再接t”的行。

```
[root@localhost ~]# grep 'r..t' /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
```

2.“*”符号

“*”符号用于匹配前一个字符0次或任意多次，比如ab*，可以匹配a、ab、abb等。“*”号经常和“.”符号加在一起使用。比如“.*”代表任意长度的不包含换行的字符。下面的例子是试图找到连续的r字母紧跟着字母t的行。由于在/etc/passwd中没有rt、rrt这样的匹配，所以该表达式实际上只找出了包含t的行（r匹配了0次）。

```
[root@localhost ~]# grep 'r*t' /etc/passwd
```

```
root:x:0:0:root:/root:/bin/bash
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
news:x:9:13:news:/etc/news:
operator:x:11:0:operator:/root:/sbin/nologin
.....(
略去内容).....
xfs:x:43:43:X Font Server:/etc/X11/fs:/sbin/nologin
```

如果把上面的‘r*t’换成‘r.*t’，代表查找包含字母r，后面紧跟任意长度的字符，再跟一个字母t的行。如下所示：

```
[root@localhost ~]# grep 'r.*t' /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
vcsa:x:69:69:virtual console memory owner:/dev:/sbin/nologin
rpc:x:32:32:Portmapper RPC user:/:/sbin/nologin
pcap:x:77:77:/:/var/arpwatch:/sbin/nologin
sshd:x:74:74:Privilege-separated
SSH:/var/empty/sshd:/sbin/nologin
avahi-autoipd:x:100:101:avahi-autoipd:/var/lib/avahi-
autoipd:/sbin/nologin
xfs:x:43:43:X Font Server:/etc/X11/fs:/sbin/nologin
```

3.“\{n,m\}”符号

虽然“*”可用于重复匹配前一个字符，但却不能精确地控制匹配的重复次数，使用“\{n,m\}”符号则能更加灵活地控制字符的重复次数，典型的有以下3种形式：

·\{n\}匹配前面的字符n次。下例匹配的是包含root的行（r和t中包含两个o）。

```
[root@localhost ~]# grep 'ro\{2\}t' /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
```

·\{n,\}匹配前面的字符至少n次以上（含n次）。

```
[root@localhost ~]# grep 'ro\{0,\}t' /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
vcsa:x:69:69:virtual console memory owner:/dev:/sbin/nologin
rpc:x:32:32:Portmapper RPC user:/:/sbin/nologin
```

·\{n,m\}匹配前面的字符n到m次。

4.“^”符号

这个符号位于键盘数字6的上面，又称尖角号。这个符号用于匹配开头的字符。比如说“^root”匹配的是以字母root开始的行。

```
[root@localhost ~]# grep '^root' /etc/passwd
root:x:0:0:root:/root:/bin/bash
```

5.“\$”符号

和上面的尖角号相对，“\$”用于匹配尾部，比如说“abc\$”代表的是以abc结尾的行。如果是“^\$”则代表该行为空，因为^和\$间什么都没有。下例匹配的是以r开头，中间有一串任意字符，以h结尾的行。

```
[root@localhost ~]# grep '^r.*h$' /etc/passwd
root:x:0:0:root:/root:/bin/bash
```

6.“[]”符号

这是一对方括号，用于匹配方括号内出现的任一字符。比如说单项选择题的答案，可能是A、B、C、D选项中的任意一种，用正则表达式表示就是[ABCD]。如果遇到比较大范围的匹配，比如说要匹配任意一个大写字母，就需要使用“-”号做范围限定，写成[A-Z]，要匹配所有字母则写成[A-Za-z]。一定要注意，这里“-”的作用不是充当一个字符。

如果是要匹配不是大写字母A、B、C、D的字符又该怎么写呢？还记得上面的“^”号吗，如果这个符号出现在[]中，则代表取反，也就是“不是”的意思。那这里的写法就是[^A-D]，事情变得有点复杂了。

这里举个例子，看如何匹配手机号。手机号是11位连续的数字，第一位一定是1，所以表示为“^1”；第二位有可能是3（移动）或8（联通），表示为“[38]”；后面连续9个任意数字，表示为“[0-9]{9}”；所以整个表达式应该写为“^1[38][0-9]{9}”。

7.“\”符号

假设有个固定电话号码021-88888888，当然也可以写成021 88888888（区号和电话号码之间用空格隔开），它们的不同之处就是区号和电话号码之间使用的符号不同，一个是“-”，一个是空格。那么，对于这个电话号码要怎么匹配呢，很容易地想到应该使用“[]”来匹配。但是这么写：[-]，对吗？答案是否定的，因为“-”放到“[]”中有特别的含义。为了表示其作为一个字符的本意，就要使用“\”符了。这个符号代表转义字符，我们可以对很多特殊的字符进行“转义”，让它只代表字符本身，因此这里的写法就应该是[\\-]

再举个例子，之前我们了解到“.”代表的是任意长度的不包含换行的重复字符。但是如果想要匹配任意长度的点号呢？这时使用转义字符就对了：“\\.”。如果想要对“\”符号进行转

义，就可以这样写：“\\”。

8.“\<”符号和“\>”符号

这两个符号分别用于界定单词的左边界和右边界。比如说“\<hello”用于匹配以“hello”开头的单词；而“hello\>”则用于匹配以“hello”结尾的单词。还可以使用它们的组合——“\<\>”用于精确匹配一个字符串。所以“\<hello\>”可精确匹配单词hello，而不是helloworld等。如下所示：

```
[root@localhost ~]# echo "hello" | grep '\<hello\>'
hello
[root@localhost ~]# echo "helloworld" | grep '\<hello\>'
[root@localhost ~]#      #
没有输出，表示匹配不成功
```

以上讲的是8种常见的元字符，还有些不太常用的字符，这些字符中有不少可以使用之前讲的8种基础的元字符来表示，所以笔者并不打算一一详细介绍，仅作一些列举和简单说明。

9.“\d”符号

匹配一个数字，等价于[0-9]，使用grep匹配这种正则表达式时可能会遇到无法匹配的问题。示例如下：

```
#123
是一个数字，用[0-9]
匹配成功
[root@localhost ~]# echo 123 | grep [0-9]
123
#
但是用这种方式却匹配不成功
[root@localhost ~]# echo 123 | grep '\d'
[root@localhost ~]#      #
```

没有输出，表示匹配不成功，为什么呢？

#

这是因为“\d

”是一种Perl

兼容模式的表达式，又称作PCRE

，要想使用这种模式的匹配符，

需要加上-P

参数

```
[root@localhost ~]# echo 123 | grep -P '\d'
```

```
123 #
```

这样就匹配成功了

10.“\b”符号

匹配单词的边界，比如“\bhello\b”可精确匹配“hello”单词。

```
[root@localhost ~]# echo "hello world" | grep '\bhello\b'
```

```
hello world
```

```
[root@localhost ~]# echo "helloworld" | grep '\bhello\b'
```

```
[root@localhost ~]# #
```

这里没有匹配

11.“\B”符号

匹配非单词的边界，比如hello\B可以匹配“helloworld”中的“hello”。

```
[root@localhost ~]# echo "helloworld" | grep 'hello\B'
```

```
helloworld
```

12.“\w”符号

匹配字母、数字和下划线，等价于[A-Za-z0-9]。

```
[root@localhost ~]# echo "a" | grep '\w'
a
[root@localhost ~]# echo "\\" | grep '\w'
[root@localhost ~]# #
这里没有匹配
```

13.“\W”符号

匹配非字母、非数字、非下划线，等价于`[^A-Za-z0-9]`。

```
[root@localhost ~]# echo "\\" | grep '\W'
\      #
匹配\
符号
```

14.“\n”符号

匹配一个换行符。

15.“\r”符号

匹配一个回车符。

16.“\t”符号

匹配一个制表符。

17.“\f”符号

匹配一个换页符。

18.“\s”符号

匹配任何空白字符。

19.“\S”符号

匹配任何非空白字符。

10.1.3 扩展的正则表达式

顾名思义，扩展的正则表达式一定是针对基础正则表达式的一些补充。实际上，扩展正则表达式比基础正则表达式多了几个重要的符号。不过要注意的是，在使用这些扩展符号时，需要使用egrep命令。

·“?”符号

“?”符号用于匹配前一个字符0次或1次，所以“ro?t”仅能匹配rot或rt。

·“+”符号

“+”符号用于匹配前一个字符1次以上，所以“ro+t”就可以匹配rot、root等。

·“|”符号

“|”符号是“或”的意思，即多种可能的罗列，彼此间是一种分支关系。比如说有些地区固定电话的区号是4位数，有些地方却是3位数，这样针对不同的区号就有不同的固定电话的表示方式，如下所示：

```
#
区号是3
位的固定电话的正则表达式方式
^0[0-9]\{2\}-[0-9]\{8\}
#
区号是4
位的固定电话的正则表达式方式
^0[0-9]\{3\}-[0-9]\{8\}
#
两种区号的固定电话号码可以如下写
^0[0-9]\{2,3\}-[0-9]\{8\}
```

```
#
使用 "|"
符号也可以，但是显然比上面的方式麻烦
^0[0-9]\{2\}-[0-9]\{8\}|^0[0-9]\{3\}-[0-9]\{8\}
```

·“()”符号

“()”符号通常需要和“|”符号联合使用，用于枚举一系列可替换的字符。比如说固定电话的区号和电话号码之间，可能用“-”符号或者用一个空格连接，用于匹配的正则表达式如下：

```
#
使用 "()"
和 "|"
定义连接符的写法
#
这样021-88888888
和0511 88888888
都可以匹配
^0[0-9]\{2,3\}(-| ) [0-9]\{8\}
#
这种写法可以换用 "[]"
符号表示
^0[0-9]\{2,3\}[\ \ -] [0-9]\{8\}
```

虽然以上这两种写法没有本质的不同，因为“()”和“|”可以和“[]”相互混用，但是在某些场景下，“()”和“|”可以做得更多，比如说像hard、hold或hood等这类开头和结尾的字母都一样的单词，要匹配这些就必须使用“()”和“|”了。如下所示：

```
#
使用 "()"
和 "|"
匹配hard
、hold
或hood
```

$h(ar|oo|ol)d$

10.1.4 通配符

或许这是你第一次听说“通配符”，但实际上你一定用过它，只是你并没有意识到。相信所有人都曾经用过Windows下的文件搜索功能。你可能某一次想找个.doc文件，但是又一时想不起该文件名和放置的位置（确实没有养成归档的好习惯），所以你决定把计算机上所有的.doc文件全部找出来，然后再进行人工挑选，于是你用“*”号来代替该文件的名称，以“.doc”作为扩展名进行第一次搜索，如图10-2所示。

实际上，通配符是一种特殊的语句，主要包含“*”号和“?”号（还有“{ }”、“^”、“!”）。主要用来模糊搜索文件，使用它替代一个或多个真正的字符，尤其是在不知道或者不确定完整的文件名时，用来匹配符合条件的文件。

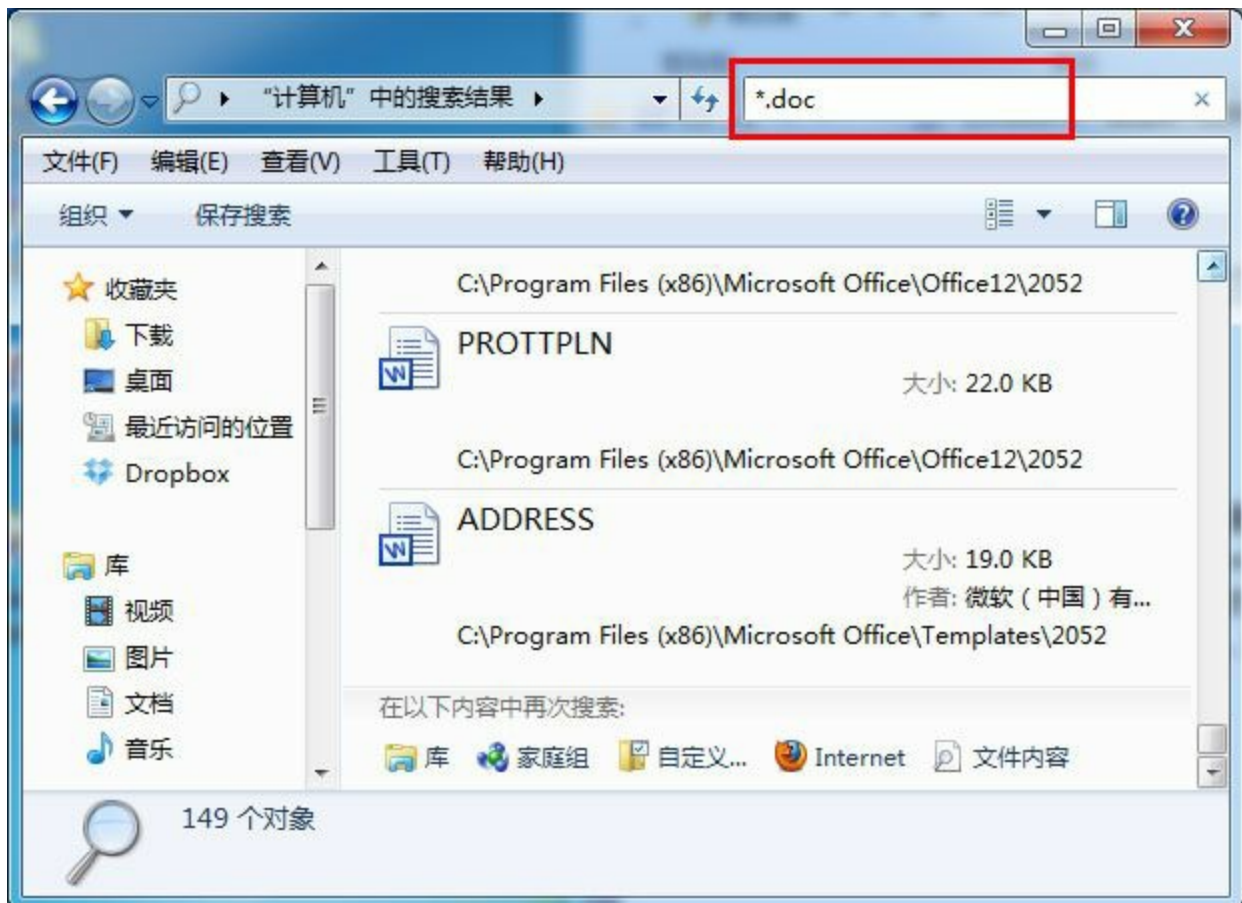


图10-2 搜索doc文件

.“*”符号

这里的“*”就是提到的第一个通配符，代表0个或多个字符。那么之前的*.doc就是指所有以.doc结尾的文件。如果想要找的文档是以字母A开头，则可用A*.doc来查找。在Linux中，列出当前目录中是否存在以.doc结尾的文件，可以使用以下命令：

```
[root@localhost ~]# ls -l *.doc
```

```
#
```

该命令执行后，shell

先要解析出命令和参数，这里的命令是ls

，参数是*.doc

，

一旦发现了*

符号，shell

就会将*.doc

解析成所有匹配的文件名，然后显示结果

.“?”符号

如果要列出以字母A开头、但是只有两个字母的文件名、以.doc结尾的文件，就需要使用“?”了。当它作为通配符使用时，代表的是任意一个字符。其写法如下：

```
[root@localhost ~]# ls -l A?.doc
```

.“{}”符号

“{}”可拥有匹配所有括号内包含的以逗号隔开的字符。例如，下面列出了所有以字母A、B、C开头，以.doc结尾的文件：

```
#
第一种方法：用“{}”
#
[root@localhost ~]# ls -l {A,B,C}.doc
#
第二种方法：用“[]”
#
[root@localhost ~]# ls -l [A-C].doc
#
以上两种方法都能满足题意，但是如果要列出以字母AB
或者CD
开头、以.doc
结尾的文件，
就只能用“{}”
了，想一想为什么
```

有意思的是，“{}”还支持嵌套的通配。以“{x, y}”为例，如果x和y各自本身也是通配符，则就变得更强大了，想一想下面例子的含义。

```
[root@localhost ~]# ls -l {[A-Z]*.doc,[0-9]??}.txt}
```

.“^”符号和“!”符号

这两个符号往往和“[]”一起使用，当出现在“[]”中的时候，代表取反。所以[^A]（或[!A]）代表不是A。

可能大家已经认识到，通配符和正则表达式之间存在的一些差异，特别是有些相同的字符既用在正则表达式中又用在通配符中，极易造成混淆和干扰，只有通过多读多想才能加深理解和认识。简要地说，正则表达式主要使用在对文件内容的匹配上，而通配符主要是用在文件名的匹配上，可以用这种方法来帮助区别二者。

10.2 正则表达式示例

在前面的第5章中，我们了解了grep的一些基本用法，但是它的功能还远远不止这些。grep的英文是Global search Regular Expression and print out the line，即全面搜索正则表达式并打印出匹配行。通过本章前面的一些实例我们也看到，grep和正则表达式结合使用后产生了强大的搜索效果。本节将通过更多的示例来介绍正则表达式和grep结合的用法，帮助大家更深入地理解和认识正则表达式和grep。提醒一下读者，由于正则表达式中含有较多特殊的字符，所以结合grep时，最好使用单引号将正则表达式括起来，以免造成错误。

为了演示grep命令的用法，首先创建一个文件RegExp.txt，文件内容如下所示：

```
[root@localhost ~]# cat RegExp.txt
----TEXT BEGIN----
good morning teacher
hello world is a script
gold sunshine looks beautiful
golden time flies
god bless me
what a delicious food
they teast Good
you fell glad
wrong word goood
wrong word gl0d
wrong word gl2d
wrong word gl3d
www.helloworld.com
www@helloworld@com
Upper case
100% means pure
php have a gd module
-----TEXT END-----
```

接下来，回顾一下grep的基本用法：

```
#
搜索含有good
单词的行
#
注意：Grep
默认是区分大小写的，所以这里只会打印出包含小写good
的行
[root@localhost ~]# grep 'good' RegExp.txt
good morning teacher
#
搜索含有good
单词的行，不区分大小写
[root@localhost ~]# grep -i 'good' RegExp.txt
good morning teacher
they teast Good
#
统计不含good
单词的行的行数，不区分大小写
[root@localhost ~]# grep -ivc 'good' RegExp.txt
19
```

下面可以正式介绍正则表达式和grep结合的用法了。

1) 使用“^”匹配行首，示例如下：

```
#
搜索以good
开头的行
[root@localhost ~]# grep '^good' RegExp.txt
good morning teacher
```

2) 使用“\$”匹配行尾，示例如下：

```
#
搜索以Good
结尾的行
```

```
[root@localhost ~]# grep 'Good$' RegExp.txt
they teast Good
```

3) 使用“^\$”组合，匹配空行，下面的命令可计算文件中共有多少行空行：

```
#
搜索空行的行数
[root@localhost ~]# grep -c '^$' RegExp.txt
2
```

4) 使用方括号匹配多种可能，示例如下：

```
#
搜索包含Good
和good
的行
[root@localhost ~]# grep '[Gg]ood' RegExp.txt
good morning teacher
they teast Good
```

5) 在方括号中使用“^”做反选，示例如下：

```
#
搜索一个包含ood
的行，但是不能是Good
或good
#
记住在方括号中使用尖角号表示的是“非”
[root@localhost ~]# grep '[^Gg]ood' RegExp.txt
what a delicious food
wrong word gooood
```

6) 使用“.”号，示例如下：

#

搜索包含一个词，该词以g
开头、紧接着是两个任意字符、再接着是一个d
的行

```
[root@localhost ~]# grep 'g..d' RegExp.txt
good morning teacher
gold sunshine looks beautiful
golden time flies
you fell glad
wrong word gl0d
wrong word gl2d
wrong word gl3d
```

#

搜索包含一个词，该词以G
或g
开头、紧接着是两个任意字符、再接着是一个d
的行

```
[root@localhost ~]# grep '[Gg]..d' RegExp.txt
good morning teacher
gold sunshine looks beautiful
golden time flies
they teast Good
you fell glad
wrong word gl0d
wrong word gl2d
wrong word gl3d
```

#

搜索这样一些行，该行包含某个单词，该词满足如下条件：

#1.

第一个字符可以是G

或g

#2.

第二个字符可以是l

或o

#3.

第三个字符可以是换行符之外的任意字符

#4.

第四个字符一定是d

```
[root@localhost ~]# grep '[Gg][lo].d' RegExp.txt
good morning teacher
gold sunshine looks beautiful
golden time flies
they teast Good
you fell glad
wrong word gl0d
wrong word gl2d
wrong word gl3d
```

7) 使用精确匹配，示例如下：

```
#
搜索含有gold
的行
#
从搜索结果中发现golden
也被匹配出来了
[root@localhost ~]# grep 'gold' RegExp.txt
gold sunshine looks beautiful
golden time flies
#
正如上例所示，一般搜索时，想要搜索含有gold
的行，发现golden
也匹配了
#
现在我们需要精确匹配含有gold
这个单词的行
[root@localhost ~]# grep '\<gold\>' RegExp.txt
gold sunshine looks beautiful
#
用"\b
"的效果和"\<\>
"一致
[root@localhost ~]# grep '\bgold\b' RegExp.txt
gold sunshine looks beautiful
```

8) 使用“*”号，示例如下：

```
#
搜索这样一些行，该行包含某个单词，该词满足如下条件：
#1.
以g
开头
#2.g
后面跟零到无限个o
#3.
零到无限个o
后面跟d
[root@localhost ~]# grep go*d RegExp.txt
```

```
good morning teacher
god bless me
wrong word goood
php have a gd module
```

9) 使用“.”号，示例如下：

```
#
搜索这样一些行，该行包含某个单词，该词满足如下条件：
#1.
以g
开头
#2.g
后面一定有字符
#3.
最后是d
[root@localhost ~]# grep 'g.*d' RegExp.txt
good morning teacher
gold sunshine looks beautiful
golden time flies
god bless me
you fell glad
wrong word goood
wrong word gl0d
wrong word gl2d
wrong word gl3d
php have a gd module
```

10) 使用“-”号，示例如下：

```
#
文件中有一些拼写错误的单词，发现是把glod
中的o
字母写成数字0
了
[root@localhost ~]# grep 'gl[0-9]d' RegExp.txt
wrong word gl0d
wrong word gl2d
wrong word gl3d
```

11) 使用“\”做字符转义，示例如下：

```
#
搜索文件中包含域名www.helloworld.com
的行
#
从搜索的结果来看，这里的“.”
“号被解析成了除换行符外的任意字符
#
想要把这个点只当作一个字符点来用，就需要对其使用转义符
[root@localhost ~]# grep 'www.helloworld.com' RegExp.txt
www.helloworld.com
www@helloworld.com
#
这里将点做转义，则输出的结果满足预期
[root@localhost ~]# grep 'www\.helloworld\.com' RegExp.txt
www.helloworld.com
```

12) 使用“\{ \}”号，示例如下：

```
#
文档中有一个单词good
被拼写错了，多写了几个o
#
搜索以字母g
开头包含两个以上o
的单词
[root@localhost ~]# grep 'go\{2,\}' RegExp.txt
good morning teacher
wrong word goood
#
搜索以字母g
开头，中间正好包含4
个o
的单词
[root@localhost ~]# grep 'go\{4\}' RegExp.txt
wrong word goood
```

13) 特殊的POSIX字符，示例如下：

#grep支持一类特殊的POSIX字符，列举如下

```
#[:alnum:]
文数字字符
#[:alpha:]
文字字符
#[:digit:]
数字字符
#[:graph:]
非空字符(
非空格、控制字符)
#[:lower:]
小写字符
#[:cntrl:]
控制字符
#[:print:]
非空字符(
包括空格)
#[:punct:]
标点符号
#[:space:]
所有空白字符(
新行，空格，制表符)
#[:upper:]
大写字符
#[:xdigit:]
十六进制数字(0-9
, a-f
, A-F)
#
搜索以大写开头的行
[root@localhost ~]# grep ^[:upper:] RegExp.txt
Upper case
#
搜索以数字开头的行
[root@localhost ~]# grep ^[:digit:] RegExp.txt
100% means pure
```

14) 使用扩展的正则表达式egrep，示例如下：

```
#
搜索g
```

和d
 之间至少有一个o
 的行
 #
 "+
 "代表匹配前面的字符1
 次以上（含1
 次）
 [root@localhost ~]# egrep 'go+d' RegExp.txt
 good morning teacher
 god bless me
 wrong word goood
 #
 搜索g
 和d
 之间只有0
 个或1
 个o
 的行（0
 次或1
 次）
 #
 "?
 "代表匹配前面的字符1
 次以上
 [root@localhost ~]# egrep 'go?d' RegExp.txt
 god bless me
 php have a gd module
 #
 搜索有glad
 或gold
 的行
 [root@localhost ~]# egrep 'glad|gold' RegExp.txt
 gold sunshine looks beautiful
 golden time flies
 you fell glad
 #
 搜索有glad
 或gold
 的行的另一种写法
 [root@localhost ~]# egrep 'g(la|ol)d' RegExp.txt
 gold sunshine looks beautiful
 golden time flies
 you fell glad

从以上的例子可以看出，正则表达式为文件行搜索提供了

强大的支持，使得搜索更为灵活，但同时也加大了使用和读写难度。要解决这个问题，只有不断地多读多用，才能较为深刻地理解正则表达式。

10.3 文本处理工具sed

10.3.1 sed介绍

sed (**s**tream **e**ditor) 是一种非交互式的流编辑器，通过多种转换修改流经它的文本。但是请注意，默认情况下，**sed**并不会改变原文件本身，而只是对流经**sed**命令的文本进行修改，并将修改后的结果打印到标准输出中（也就是屏幕）。所以本节讲的所有的**sed**操作都只是对“流”的操作，并不会改变原文件。**sed**处理文本时是以行为单位的，每处理完一行就立即打印出来，然后再处理下一行，直至全文处理结束。**sed**可做的编辑动作包括删除、查找替换、添加、插入、从其他文件中读入数据等。

注意：要想保存修改后的文件，必须使用重定向生成新的文件。如果想直接修改源文件本身则需要使用“-i”参数。

sed命令使用的场景包括以下一些：

- 常规编辑器编辑困难的文本。
- 太过于庞大的文本，使用常规编辑器难以胜任（比如说vi一个几百兆的文件）。
- 有规律的文本修改，加快文本处理速度（比如说全文替换）。

为了演示**sed**的用法，首先准备如下文件：

```
[root@localhost ~]# cat Sed.txt
this is line 1, this is First line
this is line 2, the Second line, Empty line followed
this is line 4, this is Third line
```

this is line 5, this is Fifth line

使用sed修改文件流的方式如下：

sed [options] 'command' file

#options

是sed

可以接受的参数

#command

是sed

的命令集（一共有25

个）

#

使用 -e

参数和分号连接多编辑命令

#

该参数本身只是sed

的一个简单参数，表示将下一个字符串解析为sed

编辑命令

#

一般情况下可以忽略，但是当sed

需要传递多个编辑命令时该参数就不能少了

#

下面的例子就是演示了在将this

改为That

的同时，还要将line

改为LINE

#

两个编辑命令前都要使用 -e

参数，如果有更多的编辑需求，以此类推即可

```
[root@localhost ~]# sed -e 's/this/That/g' -
```

```
e 's/line/LINE/g' Sed.txt
```

```
That is LINE 1, That is First LINE
```

```
That is LINE 2, the Second LINE, Empty LINE followed
```

```
That is LINE 4, That is Third LINE
```

```
That is LINE 5, That is Fifth LINE
```

#

使用分号连接两个都编辑命令

#

上面的命令可以用分号改写为：

```
[root@localhost ~]# sed 's/this/That/g ; s/line/LINE/g' Sed.t
```

10.3.2 删除

使用d命令可删除指定的行，示例如下：

```
#
将file
的第一行删除后输出到屏幕
#
你应该知道如何删除第二行了
[root@localhost ~]# sed '1d' Sed.txt
this is line 2, the Second line, Empty line followed
this is line 4, this is Third line
this is line 5, this is Fifth line
#
由于sed
默认不修改原文件，如果希望保存修改后的文件则需要用重定向
sed '1d' Sed.txt > saved_file
#
如果想直接修改文件，使用-i
参数
#
这里不会有任何输出，而是直接修改了源文件，删除了第一行
#Sed -i '1d' file
#
删除指定范围的行（第1
行到第5
行）
[root@localhost ~]# sed '1,3d' Sed.txt
this is line 4, this is Third line
this is line 5, this is Fifth line
#
删除指定范围的行（第一行到最后行）
[root@localhost ~]# sed '1,$d' Sed.txt
[root@localhost ~]# #
清空了Sed.txt
文件
#
删除最后一行
[root@localhost ~]# sed '$d' Sed.txt
this is line 1, this is First line
this is line 2, the Second line, Empty line followed
this is line 4, this is Third line
#
```

删除除指定范围以外的行（只保留第5行）

#

要删除其他的行请举一反三

```
[root@localhost ~]# sed '5!d' Sed.txt
```

```
this is line 5, this is Fifth line
```

#

删除所有包含Empty

的行

```
[root@localhost ~]# sed '/Empty/d' Sed.txt
```

```
this is line 1, this is First line
```

```
this is line 4, this is Third line
```

```
this is line 5, this is Fifth line
```

#

删除空行

```
[root@localhost ~]# sed '/^$/d' Sed.txt
```

```
this is line 1, this is First line
```

```
this is line 2, the Second line, Empty line followed
```

```
this is line 4, this is Third line
```

```
this is line 5, this is Fifth line
```

10.3.3 查找替换

使用s命令可将查找到的匹配文本内容替换为新的文本。

```
#s
命令用于替换文本，本例中使用LINE
替换line
#
请注意每一行只有第一个line
被替换了，默认情况下只替换第一次匹配到的内容
[root@localhost ~]# sed 's/line/LINE/' Sed.txt
this is LINE 1, this is First line
this is LINE 2, the Second line, Empty line followed
this is LINE 4, this is Third line
this is LINE 5, this is Fifth line
#
要想每行最多匹配2
个line
，并改为LINE
，可用如下方式
#
注意到第2
行中有3
个line
，前两个被替换了，第三个没有变化
[root@localhost ~]# sed 's/line/LINE/2' Sed.txt
this is line 1, this is First LINE
this is line 2, the Second LINE, Empty line followed
this is line 4, this is Third LINE
this is line 5, this is Fifth LINE
#s
命令利用g
选项，可以完成所有匹配值的替换
[root@localhost ~]# sed 's/line/LINE/g' Sed.txt
this is LINE 1, this is First LINE
this is LINE 2, the Second LINE, Empty LINE followed
this is LINE 4, this is Third LINE
this is LINE 5, this is Fifth LINE
#
只替换开头的this
为that
[root@localhost ~]# sed 's/^this/that/' Sed.txt
that is line 1, this is First line
```

that is line 2, the Second line, Empty line followed
that is line 4, this is Third line
that is line 5, this is Fifth line

10.3.4 字符转换

使用y命令可进行字符转换，其作用为将一系列字符逐个地变换为另外一系列字符，基本用法如下：

```
#
该命令会将file
中的O
转换为N
、L
转换为E
、D
转换为W
#
注意转换字符和被转换字符的长度要相等，否则sed
无法执行
sed 'y/OLD/NEW/' file
```

下面的命令演示了将数字1转换为A，2转换为B，4转换为C，5转换成D的用法。

```
[root@localhost ~]# sed 'y/1245/ABCD/' Sed.txt
this is line A, this is First line
this is line B, the Second line, Empty line followed
this is line C, this is Third line
this is line D, this is Fifth line
```

10.3.5 插入文本

使用*i*或*a*命令插入文本，其中*i*代表在匹配行之前插入，而*a*代表在匹配行之后插入，示例如下：

```
#
使用i
在第二行前插入文本
[root@localhost ~]# sed '2 i Insert' Sed.txt
this is line 1, this is First line
Insert
this is line 2, the Second line, Empty line followed
this is line 4, this is Third line
this is line 5, this is Fifth line
#
使用a
在第二行后插入文本
[root@localhost ~]# sed '2 a Insert' Sed.txt
this is line 1, this is First line
this is line 2, the Second line, Empty line followed
Insert
this is line 4, this is Third line
this is line 5, this is Fifth line
#
在匹配行的上一行插入问题
[root@localhost ~]# sed '/Second/i\Insert' Sed.txt
this is line 1, this is First line
Insert
this is line 2, the Second line, Empty line followed
this is line 4, this is Third line
this is line 5, this is Fifth line
```

10.3.6 读入文本

使用r命令可从其他文件中读取文本，并插入匹配行之后，示例如下：

```
#
将/etc/passwd
中的内容读出放到Sed.txt
空行之后
[root@localhost ~]# sed '/^$/r /etc/passwd' Sed.txt
this is line 1, this is First line
this is line 2, the Second line, Empty line followed
root:x:0:0:root:/root:/bin/bash
.....(
略去内容).....
apache:x:48:48:Apache:/var/www:/sbin/nologin
this is line 4, this is Third line
this is line 5, this is Fifth line
```

10.3.7 打印

使用p命令可进行打印，这里使用sed命令时一定要加-n参数，表示不打印没关系的行。从之前的例子中可以看出，由于sed的工作原理是基于行的，因此每次都有大量的输出。可是这些输出中有一些是我们并不需要看到的，而只需要输出匹配的行或者处理过的行就好了。简单来说，打印操作是删除操作的“逆操作”。下面使用演示来具体说明：

```
#
打印出文件中指定的行
[root@localhost ~]# sed -n '1p' Sed.txt
this is line 1, this is First line
#
将the
替换成THE
#sed
实际处理了第二行，其他几行由于没有匹配所以并未真正处理
#
但是sed
的工作原理是基于流的，所以所有流过的行都打印出来了
[root@localhost ~]# sed 's/the/THE/' Sed.txt
this is line 1, this is First line
this is line 2, THE Second line, Empty line followed
this is line 4, this is Third line
this is line 5, this is Fifth line
#
使用p
命令，则只打印实际处理过的行，简化了输出（使用-n
参数）
[root@localhost ~]# sed -n 's/the/THE/p' Sed.txt
this is line 2, THE Second line, Empty line followed
```

10.3.8 写文件

正如之前所说，`sed`本身默认并不改写原文件，而只是对缓冲区的文本做了修改并输出到屏幕。所以想保存文件，除了之前提到的两种方法外（使用重定向或`-i`参数），还可以使用`w`命令将结果保存到外部指定文件。示例如下：

```
[root@localhost ~]# sed -n '1,2 w output' Sed.txt
[root@localhost ~]# #
这里没有任何输出，因为输出被重定向到文件了
#
文件output
中的内容正是Sed.txt
文件中前两行的内容
[root@localhost ~]# cat output
this is line 1, this is First line
this is line 2, the Second line, Empty line followed
```

10.3.9 sed脚本

在平日的工作中，我们可能需要定期对一些文件做分析操作，这种例行工作往往有一定“标准化”的操作，比如说先去除文件中所有的空行，然后再全部替换某些字符等，这种过程类似于生产线上程式化的流水作业。事实上，可以把这些动作静态化地写到某个文件中，然后调用sed命令并使用-f参数指定该文件，这样就可以将这一系列动作“装载”并应用于指定文件中，这无疑加快了工作效率，这种文件就是sed脚本。请观察下面的sed脚本：

```
#
该sed
脚本的作用是将全文的this
改为THAT
，并删除所有空行
[root@localhost ~]# cat Sed.rules
s/this/THAT/g
/^$/d
#
使用 -f
参数指定该脚本并应用于Sed.txt
#
从输出内容中可以看出执行效果
[root@localhost ~]# sed -f Sed.rules Sed.txt
THAT is line 1, THAT is First line
THAT is line 2, the Second line, Empty line followed
THAT is line 4, THAT is Third line
THAT is line 5, THAT is Fifth line
```

10.3.10 高级替换

上面介绍了几个常用的sed命令，并逐个进行了演示。但是在实践中，往往由于需求复杂而无法简单满足。这里将介绍一些不太常用的高级编辑命令供读者参考。

·替换匹配行的下一行

想要修改匹配行的下一行的文本，就需要使用n命令了。该命令的作用在于读取匹配行的下一行，然后再用n命令后的编辑指令来处理该行。在下面的Sed.txt文件中有一行空白行，现在想将该空格行的下一行中的line改为LINE，而文本中其他部分保持不变，操作如下：

```
[root@localhost ~]# sed '/^$/ {n;s/line/LINE/g}' Sed.txt
this is line 1, this is First line
this is line 2, the Second line, Empty line followed
this is LINE 4, this is Third LINE
this is line 5, this is Fifth line
```

·将模式空间的内容放入存储空间以便于接下来的编辑

实现该功能，就要引入H/h/G/g这4个命令了，这几个命令都是用于模式空间和存储空间转换的。首先来解释一下两个新出现的词的含义。

模式空间：当前输入行的缓冲区。

存储空间：模式空间以外的一个预留缓冲区。

下面来看看H/h/G/g这4个命令的具体含义：

·H：将模式空间的内容追加到存储空间。

·h: 将模式空间的内容复制到存储空间，覆盖原有存储空间。

·G: 将存储空间的内容追加到模式空间。

·g: 将存储空间的内容复制到模式空间。

以下是相应的示例：

```
#Sed2.txt
文件内容
[root@localhost ~]# cat Sed2.txt
a
b
aa
bb
#
实现第一行与第二行以及第三行与第四行的反转
[root@e-bai ~]# sed '/a/{h;d};/b/G' Sed2.txt
b
a
bb
aa
```

10.3.11 sed总结

sed命令的功能十分强大，想面面俱到地精细列举确实不太可能。事实上，由于sed本身的复杂度，以及和正则表达式的结合，使得sed命令非常难以掌握。要想更全面地了解sed建议读者系统地阅读sed的相关书籍。限于篇幅，笔者选取了在实际工作中大量使用到的功能，并做了对应的演示，希望读者能将所有示例亲自动手做一遍，一定能帮助自己提高理解。初学者在读完本章节后，可能会忘记绝大部分内容，只有靠不断使用、加深记忆才能逐渐了解和掌握。表10-1～表10-3总结了绝大部分sed的命令和作用，方便读者查询。

表10-1 sed常用的命令

sed 命令	作 用
a	在匹配行后面加入文本
c	字符转换
d	删除行
D	删除第一行
i	在匹配行前面加入文本
h	复制模板块的内容到存储空间
H	追加模板块的内容到存储空间
g	将存储空间的内容复制到模式空间
G	将存储空间的内容追加到模式空间
n	读取下一个输入行，用下一个命令处理新的行
N	追加下一个输入行到模板块后并在二者间插入新行

(续)

sed 命令	作 用
p	打印匹配的行
P	打印匹配的第一行
q	退出 sed
r	从外部文件中读取文本
w	追加写文件
!	匹配的逆
s/old/new	用 new 替换正则表达式 old
=	打印当前行号

表10-2 sed常用的参数

sed 参数	作 用
-e	多条件编辑
-h	帮助信息
-n	不输出不匹配的行
-f	指定 sed 脚本
-V	版本信息
-i	直接修改原文件

表10-3 sed常用的正则表达式匹配

元字符	作 用
<code>^</code>	匹配行的开始。如： <code>/^cat/</code> 匹配所有以 <code>cat</code> 开头的行
<code>\$</code>	匹配行的结束。如： <code>/cat\$/</code> 匹配所有以 <code>cat</code> 结尾的行
<code>.</code>	匹配任一非换行字符。如： <code>/c.t/</code> 匹配 <code>c</code> 后接一个任意字符，然后是 <code>t</code>
<code>*</code>	匹配零个或任意多个字符。如： <code>/*cat/</code> 匹配一串字符后紧跟 <code>cat</code> 的所有行
<code>[]</code>	匹配指定范围内的字符。如： <code>/[Cc]at/</code> 匹配 <code>cat</code> 和 <code>Cat</code>
<code>[^]</code>	匹配不在指定范围内的字符。如： <code>/[^A-Z]/</code> 匹配不是以大写字母开头的行
<code>\(..)</code>	保存匹配的字符。如： <code>s/(love)able\lrs/</code> , <code>loveable</code> 被替换成 <code>lovers</code>
<code>&</code>	保存搜索字符用来替换其他字符。如 <code>s/love/**&*/</code> , <code>love</code> 变成 <code>**love**</code>
<code>\<</code>	锚定单词的开始。如： <code>/^<cat/</code> 匹配包含以 <code>cat</code> 开头的单词的行
<code>\></code>	锚定单词的结束。如： <code>/cat>/</code> 匹配包含以 <code>cat</code> 结尾的单词的行
<code>x\{n\}</code>	重复字符 <code>x</code> , <code>m</code> 次。如： <code>/o\{5\}/</code> 匹配包含 5 个 <code>o</code> 的行
<code>x\{m,\}</code>	重复字符 <code>x</code> , 至少 <code>m</code> 次。如： <code>/o\{5,\}/</code> 匹配至少有 5 个 <code>o</code> 的行
<code>x\{n,m\}</code>	重复字符 <code>x</code> , 至少 <code>m</code> 次, 不多于 <code>n</code> 次。如： <code>/o\{5,10\}/</code> 匹配 5 到 10 个 <code>o</code> 的行

10.4 文本处理工具awk

上一节中介绍的sed其实是以行为单位的文本处理工具，而awk则是基于列的文本处理工具，它的工作方式是按行读取文本并视为一条记录，每条记录以字段分割成若干字段，然后输出各字段的值。事实上，awk是一种编程语言，其语法异常复杂，所以awk也是一种较难掌握的文本处理工具。本节将使用大量的例子来直接演示awk的常见用法，让读者能迅速学会使用。

awk认为文件都是结构化的，也就是说都是由单词和各种空白字符组成的，这里的“空白字符”包括空格、Tab，以及连续的空格和Tab等。每个非空白的部分叫做“域”，从左到右依次是第一个域、第二个域，等等。\$1、\$2分别用于表示域，\$0则表示全部域。

为了演示awk的用法，首先创建文件Awk.txt，文件内容如下所示：

```
[root@localhost ~]# cat Awd.txt
john.wang      Male      30        021-11111111
lucy.yang      Female    25        021-22222222
jack.chen      Male      35        021-33333333
lily.gong      Female    20        021-44444444      ShangHai
```

10.4.1 打印指定域

既然awk使用\$1、\$2代表不同的域，则可以打印指定域。拿Awk.txt的第一行来说，第一个域为john.wang，第二个域为Male，第三个域为30，第四个域为021-11111111。在下面的演示中，第一条命令打印了\$1、\$4这两个域，而第二条命令则打印了全部的域。

```
#
只打印姓名和电话号码
[root@localhost ~]# awk '{print $1, $4}' Awd.txt
john.wang 021-11111111
lucy.yang 021-22222222
jack.chen 021-33333333
lily.gong 021-44444444
#
打印全部内容
[root@localhost ~]# awk '{print $0}' Awd.txt
john.wang      Male      30      021-11111111
lucy.yang      Female    25      021-22222222
jack.chen      Male      35      021-33333333
lily.gong      Female    20      021-44444444      ShangHai
```

10.4.2 指定打印分隔符

默认情况下awk是使用空白字符作为分隔符的，但是也可以通过-F参数指定分隔符，来区分不同的域（有点像之前学过的cut命令）。示例如下：

```
#
指定".
"作为分隔符，这样每一行的$1
就是".
"之前的字符，$2
就是".
"之后的字符
比如说第一行的$1
是"john
", $2
是"Male      30      021-11111111
"

[root@localhost ~]# awk -F. '{print $1, $2}' Awd.txt
john wang      Male      30      021-11111111
lucy yang      Female    25      021-22222222
jack chen      Male      35      021-33333333
lily gong      Female    20      021-44444444      ShangHai
```

10.4.3 内部变量NF

文件Awd.txt所包含的内容并不多，所以我们很容易地知道它的前3行中每行都有4个域，而最后一行是5个域。但是如果有时候文件很大，每行列数都不一样，靠观察就不现实了，必须通过特定的方式来获得文件的列数。通过awk的内部变量NF可以简单地做到这点。当然，如果你指定了不同的分隔符，结果可能不一样。示例如下：

```
#
使用默认分隔符
[root@localhost ~]# awk '{print NF}' Awd.txt
4
4
4
5
#
指定分隔符
[root@localhost ~]# awk -F. '{print NF}' Awd.txt
2
2
2
2
```

10.4.4 打印固定域

通过内部变量可以简单地得到每行的列数，而如果在NF之前加上\$符号，则代表“最后一列”，这样不管每行有多少列，只要使用\$NF就能打印出最后一行。如果是倒数第二行呢？读者可以先思考一下再看示例。

```
#
打印最后一行
[root@localhost ~]# awk '{print $NF}' Awd.txt
021-11111111
021-22222222
021-33333333
ShangHai
#
用$(NF-1)
打印倒数第二行
[root@localhost ~]# awk '{print $(NF-1)}' Awd.txt
30
25
35
021-44444444
```

10.4.5 截取字符串

可以使用`substr()`函数对指定域截取字符串，该函数的基本使用方法如下：

```
substr(  
指定域,  
第一个开始字符的位置,  
第二个结束的位置)  
#  
其中第二个结束的位置可以为空，这样默认输出到该域的最后一个字符
```

下例中将输出`Awk.txt`文件第一个域的第六个字符到最后一个字符的内容：

```
#  
该例中，第二个结束位置省略，所以结束位置为第一个域的最后一个字符  
[root@localhost ~]# cat Awd.txt | awk '{print substr($1,6)}'  
wang  
yang  
chen  
gong
```

10.4.6 确定字符串的长度

使用内部变量`length`可以确定字符串的长度，示例如下：

```
[root@localhost ~]# cat Awd.txt | awk '{print length}'
30
32
30
41
```

10.4.7 使用awk求列和

结构化的数据在系统中是随处可见的，比如用ls-l命令得到的输出、各类系统日志等。在日常工作中，经常有将其中的数据进行相加的需求。下面演示了对所有人的年龄进行的一些计算。请注意，年龄字段是第三个域：

```
#
求年龄的和
[root@localhost ~]# cat Awd.txt | awk 'BEGIN{total=0}
{total+=$3}END{print total}'
110
#
求平均年龄
[root@localhost ~]# cat Awd.txt | awk 'BEGIN{total=0}
{total+=$3}END{print total/NR}'
27.5
```

第11章 Shell编程概述

11.1 Shell简介

11.1.1 Shell是什么

前面的10个章节全面介绍了Linux系统的基础知识和常见命令。从本章开始，将把重点转到Shell编程上。实际上，读者很可能已经不止一次地接触到了Shell，只是没有注意而已：当你使用ssh客户端工具远程连接到系统，或坐在一台服务器前输入密码登录到系统中时，面前呈现的跳动的光标就是一个Shell。在计算机语言中，Shell是指一种命令行解释器，是为用户和操作系统之间通信提供的一种接口（想象一下，如果没有一种与计算机沟通的方式，那么计算机如何得到来自人脑的指令呢），它接受来自用户输入的命令，并将其转换为一系列的系统调用送到内核执行，并将结果输出给用户。图11-1显示了Shell在操作系统中的位置。

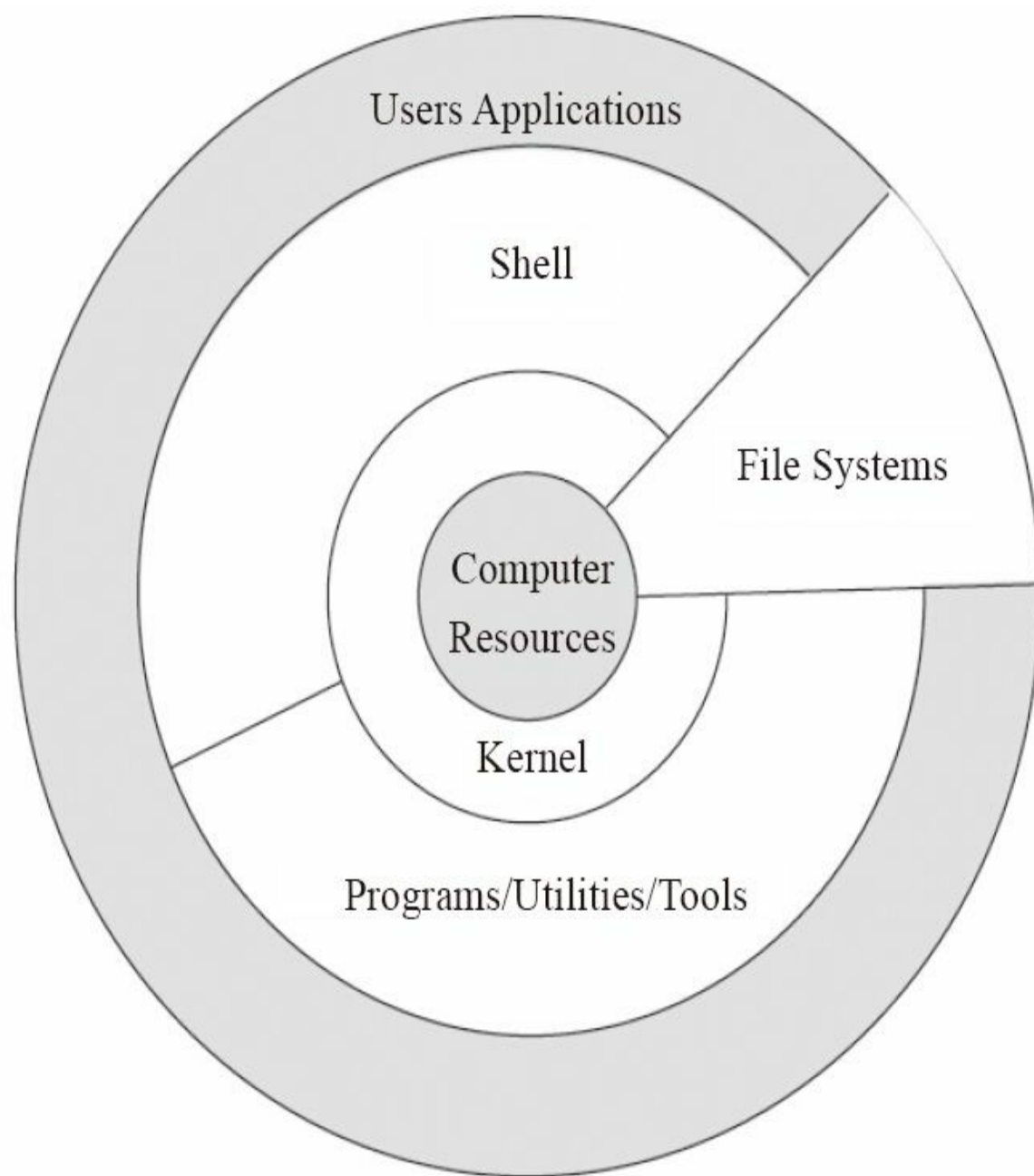


图11-1 Shell在系统中的位置

Shell分为两大类，一类是图形界面Shell（Graphical User Interface），用户可以在GNOME桌面空白处单击鼠标右键，在弹出的菜单中单击Open Terminal选项，打开Shell，如图11-2和图11-3所示。另一类是命令行式Shell（Command Line Interface），即CLI，如图11-4所示。



图11-2 在图形桌面打开终端

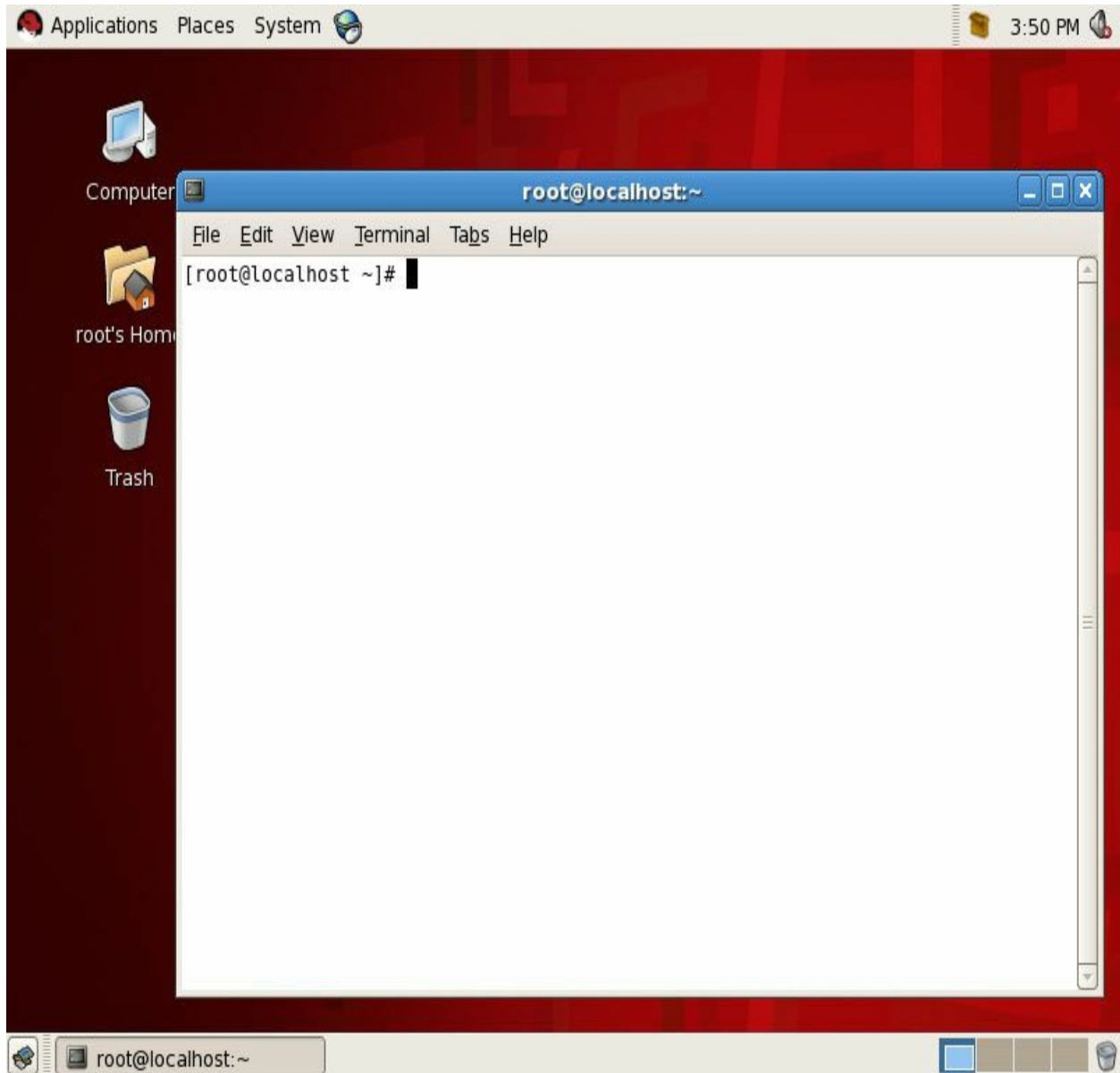


图11-3 图形模式下的终端

```
Red Hat Enterprise Linux Server release 5.5 (Tikanga)
Kernel 2.6.18-194.el5 on an i686

localhost login: root
Password:
Last login: Mon Mar 11 23:37:34 on tty1
[root@localhost ~]# _
```

图11-4 命令行模式下的终端

事实上，Shell不只是一种解释器（在用户和系统间起着桥梁的作用），还是一种编程工具，称为脚本语言。与编译型语言（比如C/C++/Java等）不同，脚本语言又被称作解释型语言，这种语言经过编写后不需要做任何编译就可以运行。

什么是解释型语言呢？这就要说到计算机运行程序的两种方式了。计算机不能理解高级语言，只能理解机器语言，所以必须把高级语言翻译为机器码。而这种翻译的方式有两类，一类是编译，一类是解释，不同之处在于翻译的时间不同。编译型语言是运行前翻译，一般是使用编译工具将程序源码处理成机器认识的可执行文件（比如说Windows下的exe文件，Linux下的二进制可执行性文件），这种文件一旦产生，以后运行时将不需要再次翻译，所以一般来说，编译型语言的效率较高；而解释型语言是运行时翻译，执行一条语句就立即翻译一条，而且每次执行程序都需要进行解释，相对来说效率较低。但是也不能简单地认为编译型语言就一定比解释型效率高，随着解释器的发展，部分解释器能在运行程序时动态优化代码，因此这种效率差距也在一定程度上不断减小。

目前RedHat和CentOS发行版中默认安装了多种Shell，可以使用如下命令来确认系统中可用的Shell是什么版本：

```
[root@localhost ~] # cat /etc/Shells
/bin/sh #Bourne Shell
/bin/bash      #Bourne again Shell
/sbin/nologin  #
非登录Shell
/bin/tcsh      #tC Shell
/bin/csh       #C Shell
/bin/ksh       #Korn Shell
```

用户可以根据自己的偏好和擅长选择使用Shell。目前众多的Linux系统默认采用Bash Shell，所以本书只涉及Bash Shell的讲解，后面如果不做特别说明，书中所提的Shell都是指Bash Shell。

11.1.2 Shell的历史

Bourne Shell是第一个重要的Shell，发布于1977年，作为UNIX 7的默认Shell，在系统中的位置是/bin/sh。目前大多数的UNIX和Linux系统还保留着这个/bin/sh，或者将其连接到其他Shell上，比如RedHat和CentOS就是将这个文件作为一个连接文件连接到Bash Shell上：

```
[root@localhost ~]# ls -l /bin/sh
lrwxrwxrwx 1 root root 4 Feb 26 21:14 /bin/sh -> bash
```

由于Bourne Shell一直没有正式的版本号，而且由于交互性不够友好等原因，导致了后来C Shell的出现。C Shell诞生于20世纪70年代，由当时加州大学伯克利分校的一名学生编写。该Shell以其语法和C类似而得名，它有着良好的交互性和较快的执行速度。但是缺点是不支持正则表达式，所以一直也没能被UNIX广泛使用，所以没有大范围的流行。

20世纪80年代初出现了Korn Shell，它不但向后兼容Bourne Shell，同时又汲取C Shell的优点，早期版本为ksh88，后来又发布了ksh93版本，成为AIX4上的默认Shell。

20世纪80年代末出现了Bash Shell，其完全兼容Bourne Shell，而且意图也非常明显：就是要代替Bourne Shell。首次发布于1989年，并作为GNU项目免费公布使用，后广泛应用于各类UNIX和Linux发行版中。实际上，Bash是Bourne Again Shell的简称，不过，它在兼容Bourne Shell的同时又加入了很多新功能，并从其他Shell中借用了不少好的功能，可以说是众多Shell的集大成者。目前最新的版本是由GNU于2011年公布的Bash 4.2，但是本文将以3.2版本为主来进行讲解，因为这是目前大多数的Linux发行版使用的版本。

#RedHat5.5

使用的bash

版本为3.2

[root@localhost ~]# bash --version

GNU bash, version 3.2.25(1)-release (i686-redhat-linux-gnu)

Copyright (C) 2005 Free Software Foundation, Inc.

11.1.3 Shell的功能

当一台系统运行起来时，内核（kernel）会被调入内存中运行，由内核执行所有底层的工作，它会将所有应用程序及用户的操作翻译成CPU的基本指令，并将其送至处理器。这些过程听起来非常复杂，而且实际上也确实是非常底层和技术化的。为了对用户屏蔽这些复杂的技术细节，同时也是为了保护内核不会因用户直接操作而受到损害，有必要在内核之上创建一个层，该层就是一个“壳”，也就是Shell名称的由来。

Bash Shell有两种工作模式，分别是互动模式和脚本模式。所谓互动模式就是由系统管理人员直接通过键盘输入命令，并等待其执行完毕后再执行下一条命令；而另一种模式是设计出一个脚本文件，将所有需要执行的命令写在该文件中，由Bash Shell读取并执行。很明显，后一种工作模式的效率更高，因为可以让工作变得“自动化”，这也是我们学习Shell最重要的原因——将一切工作都自动化处理。当然，在此过程中，有些脚本是需要一定情况下才能确保运行成功的，为此必须加入更多的判断功能。除此以外，我们还可能需要使用循环来运行一些需要重复执行的任务，等等。这些需求使得Shell实际上已经发展成为一种开发工具。Shell入门并不难，但是要想学精却需要经过大量阅读、使用、出错并从错误中总结的过程，这样才能不断提高对其的掌控能力。

11.1.4 Shell编程的优势

首先，Shell是一门非常容易入门的语言，语法结构简单。初学者学习Shell编程也可以非常迅速地上手，如果是有过其他编程经验的读者，往往只需要几小时便可掌握Shell语法。其次，Shell命令很简单，即便遇到不清楚用法的地方，借助Linux系统中强大的帮助机制也可以立刻查到用法。再次，Shell是解释型语言，用前不需要编译，可以一边开发一边测试，所以开发效率非常高。最后，Shell具备一定的跨平台性，使用POSIX所定义的规范，可以做到脚本无须修改就能在不同的系统中运行。

11.2 第一个Shell脚本

11.2.1 编辑第一个Shell脚本

几乎所有编程语言的教程都是从使用著名的“Hello World”开始的，出于对这种传统的尊重（或者说落入俗套），第一个脚本我们命名为HelloWorld.sh。下面创建该文件，编辑并保存相应的内容。

```
[root@localhost ~]# cat HelloWorld.sh
#!/bin/bash
#This Line is a comment
echo "Hello World"
```

好了，现在我们已经完成了一个程序了，不难吧？我承认这确实太简单了点，但是不可否认的是，这确实是一个可以工作的脚本，虽然只有两行。现在来解释一下这个脚本。一个Shell脚本永远是以“#!”开头的，这是一个脚本开始的标记，它是在告诉系统执行这个文件需要使用某个解释器，后面的/bin/bash就是指明了解释器的具体位置。

第二行同样是以“#”开头的，但是这里是一个注解。脚本中所有以“#”开头的都是注解（当然以“#!”开头的除外）。写脚本的时候，多写注解是非常有必要的，以方便其他人能看懂你的脚本，也方便后期自己维护时看懂自己的脚本——实际上，即便是自己写的脚本，在经过一段时间后也很容易忘记。

第三行是一句非常简单的命令：输出“Hello World”。其实这条命令与在终端中执行的效果是一样的——最简单的脚本就是命令的罗列。

11.2.2 运行脚本

有几种方式来运行上面的HelloWorld.sh，第一种就是在该脚本所在的目录中直接bash这个脚本。实际上，如果使用这种方式来运行脚本，该脚本中的第一行“#!/bin/bash”就可以不需要了，因为直接bash一个文件就是指定了使用Bash Shell来解释脚本内容。

```
[root@localhost ~]# bash HelloWorld.sh
Hello World
```

第二种方式是给该脚本加上可执行权限，然后使用“./”来运行，它代表运行的是当前目录下的HelloWorld.sh脚本，如果采用这种方式而脚本没有可执行权限则会报错。

```
[root@localhost ~]# chmod +x HelloWorld.sh
[root@localhost ~]# ./HelloWorld.sh
Hello World
#
如果脚本没有可执行权限，则会报权限错误
[root@localhost ~]# ./HelloWorld.sh
-bash: ./HelloWorld.sh: Permission denied
```

如果希望该脚本能成为默认的系统命令，简单地将该脚本复制到任一系统\$PATH变量所包含的目录中，同时赋予可执行权限，下次运行的时候只需要直接输入该命令即可。也支持用Tab键补全命令。下例就是将其复制到了/bin目录，并执行该脚本的情况。

```
[root@localhost ~]# chmod +x HelloWorld.sh
[root@localhost ~]# mv HelloWorld.sh /bin/
[root@localhost ~]# HelloWorld.sh
```

Hello World

11.2.3 Shell脚本的排错

Shell脚本在执行时出错是很常见的，最简单的原因无外乎脚本在编写的过程中出现了语法错误或者不小心输错了命令等。找出脚本中的错误是很重要的能力。假设在之前的脚本中，不小心把echo命令写成了ehco，那么执行的效果如下：

```
[root@localhost ~]# bash HelloWorld.sh
HelloWorld.sh: line 2: ehco: command not found
```

从报错信息很容易了解到，出错的原因是“命令不存在”。重新编辑这个文件修改成echo就可以了。如果只是语法上的错误或命令错误还是比较容易辨别的，但往往一些逻辑或算法错误不容易发现，因为它的语法正确且本身并不会造成程序运行出错。比如说下面的脚本，本来是想连续10次做某些操作的，结果却迟迟没有任何输出。仔细观察一下就知道，其实是陷入了死循环。

```
[root@localhost ~]# cat BadLoop.sh
#!/bin/bash
for ((i=10;i>0;i=i+1))
do
    #do something here
done
```

如果在循环中间的代码块中加入了echo语句，那就容易发现问题了。也就是说，在排查错误时，使用echo命令有助于观察代码执行的情况。假设在上面的脚本中使用了echo，再次运行脚本时会注意到脚本在不断地打印输出，这样就可以发现异常了。如果遇到确实需要执行很多次循环的代码段，由于每次循环的过程都很快，可能来不及观察就进入了下一次循环，那

么可以加入sleep命令降低单次循环的速度，比如使用sleep 2这样的命令，单次循环至少会耗时两秒，这就有足够的时间观察循环的过程了。

```
#!/bin/bash
for ((i=10;i>0;i=i+1))
do
    #do something here
    echo "i=$i";
    sleep 2
done
```

为了更清晰地看到脚本运行的过程，还可以借助-x参数来观察脚本的运行情况。比如在下面的代码段中，从输出可以看到变量i的值一直在向上增加，永远满足x>0的条件，所以这是一个死循环。只需要将原来代码中的i=i+1改成i=i-1就可以了。

```
[root@localhost ~]# bash -x BadLoop.sh
+ (( i=10 ))
+ (( i>0 ))
+ echo i=10
i=10
+ sleep 2
+ (( i=i+1 ))
+ (( i>0 ))
+ echo i=11
i=11
+ sleep 2
+ (( i=i+1 ))
+ (( i>0 ))
+ echo i=12
i=12
+ sleep 2
[Ctrl+c]
停止脚本
```

Shell本身并没有提供更好的排错工具，想要尽量减少错误，除了多学习它的语法，多写、多想、多改之外并没有更好的办法，只有勤加练习才能更快地进步。

为了更精细地调试运行Shell，我们可以借助第三方工具bashdb。这是一个类似于GDB的脚本调试软件，小巧而强大，具有设置断点、单步执行、观察变量等功能。读者可以到这个页面下载使用：http://sourceforge.jp/projects/sfnet_bashdb，请注意最新的版本（笔者在写本书时为4.2.-0.8，支持的Shell必须是4.2，如果你使用的是3.2版本的Bash，则请使用3.1版本的bashdb）。

具体下载安装的过程如下：

```
#
第一步：下载bashdb
（笔者使用的版本是3.1
）
[root@localhost ~]# wget \
http://nchc.dl.sourceforge.net/project/bashdb/bashdb/3.1-
0.09/bashdb-3.1-0.09.tar.gz
--2013-10-10 07:15:13--
http://nchc.dl.sourceforge.net/project/bashdb/bashdb/3.1-
0.09/bashdb-3.1-0.09.tar.gz
Resolving nchc.dl.sourceforge.net... 211.79.60.17, 2001:e10:f
Connecting to nchc.dl.sourceforge.net|211.79.60.17|:80... con
HTTP request sent, awaiting response... 200 OK
Length: 659032 (644K) [application/x-gzip]
Saving to: `bashdb-3.1-0.09.tar.gz'
100%
[=====>] 659,032      71.1K/s
2013-10-10    07:15:22    (69.9    KB/s)    -    `bashdb-3.1-
0.09.tar.gz' saved [659032/659032]
#
第二步：解压并进入解压后的目录
[root@localhost ~]# tar zxvf bashdb-3.1-0.09.tar.gz
[root@localhost ~]# cd bashdb-3.1-0.09
#
第三步：配置
[root@localhost bashdb-3.1-0.09]# ./configure
```

```

checking for emacs... emacs
checking where .elc files should go...
${datarootdir}/emacs/site-lisp
checking whether to enable maintainer-
specific portions of Makefiles... no
checking for a BSD-compatible install... /usr/bin/install -c
.....(
略去内容).....
config.status: executing default commands
=====
Bash version: GNU bash, version 3.2.25(1)-release (x86_64-
redhat-linux-gnu)
Copyright (C) 2005 Free Software Foundation, Inc.
Location: /bin/bash
We will not try to build the readarray builtin to speed up lo
#
第四步：编译并安装
[root@localhost bashdb-3.1-0.09]# make && make install
make all-recursive
make[1]: Entering directory `/root/bashdb-3.1-0.09'
Making all in test
make[2]: Entering directory `/root/bashdb-3.1-0.09/test'
make[2]: Nothing to be done for `all'.
make[2]: Leaving directory `/root/bashdb-3.1-0.09/test'
Making all in doc
.....(
略去内容).....
make[4]: Leaving directory `/root/bashdb-3.1-0.09'
test -d /usr/local/share/bashdb || /bin/mkdir -
p /usr/local/share/bashdb
make[3]: Leaving directory `/root/bashdb-3.1-0.09'
make[2]: Leaving directory `/root/bashdb-3.1-0.09'
make[1]: Leaving directory `/root/bashdb-3.1-0.09'

```

一旦安装完成，系统中便有了bashdb命令，该命令典型的使用法如下：

```

[root@localhost ~]# bashdb --debug
脚本名

```

更多命令可以在调试模式中使用，按照功能可划分为查看源代码相关功能、调试相关功能、控制相关功能三大部分，如

表11-1所示，也可以使用help命令调出帮助信息。

表11-1 bashdb的命令列表

命 令	功 能
l	打印源代码
-	打印当前执行行往前 10 行的代码
.	打印当前执行行的代码
w	打印当前执行行前后各 5 行的代码
/pattern/	往后查找匹配 pattern 的行
?pattern?	往前查找匹配 pattern 的行
H	打印历史命令
quit	退出 bashdb
info a	查看参数信息
info b	查看断点信息
info d	查看由 display 命令设置的列表
info files	查看源文件信息
info function	查看函数信息
info l	查看当前行和文件名
info p	查看当前调试状态

info stack	查看栈信息
info terminal	查看终端信息
info v	查看所有变量信息
hi n	执行第 n 条历史命令
show	显示设置信息
set	设置
e	当前环境中执行 Shell 命令
disp	设置 Shell 命令列表
step n	前进 n 步，遇到函数进入
next n	前进 n 步，遇到函数不进入
回车	重复上一个 n 或 s 命令
continue	继续执行，可以接一个数字作为参数，执行到指定行

(续)

命 令	功 能
break	后接一个数字作为参数，可设置指定行为断点
delete	后接一个断点号，用于删除指定断点
D	删除所有断点
skip n	跳过下面的 n 条语句不执行
clear	清除指定行上的所有断点
R	重启当前调试脚本
return	中断当前函数并返回
print	打印指定的变量值
help	打印帮助信息

11.3 Shell的内建命令

所谓Shell内建命令，就是由Bash自身提供的命令，而不是文件系统中的某个可执行文件。例如，用于进入或者切换目录的cd命令，虽然我们一直在使用它，但如果不加以注意很难意识到它与普通命令的性质是不一样的：该命令并不是某个外部文件，只要在Shell中你就一定可以运行这个命令。打个比方，就像使用语言互相沟通是人类与生俱来的能力，但是我们有时却需要使用移动电话来进行远距离的沟通，那么人类本身具备的语言能力就是“内建”的能力，而移动电话却是一个外部的工具。

```
#cd
命令并不是一个可执行文件
[root@localhost ~]# which cd
/usr/bin/which: no cd in (/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/root/bin)
#more
命令是一个可执行文件，文件位置为/bin/more
[root@localhost ~]# which more
/bin/more
```

还记得系统变量\$PATH吗？在第8章中介绍过，\$PATH变量包含的目录中几乎聚集了系统中绝大多数的可执行命令。通常来说，内建命令会比外部命令执行得更快，执行外部命令时不但会触发磁盘I/O，还需要fork出一个单独的进程来执行，执行完成后再退出。而执行内建命令相当于调用当前Shell进程的一个函数。

Shell的内建命令众多，在3.2.25版本的Bash中有几十个，如下所示：

```
bash, :, ., [, alias, bg, bind, break, builtin, cd, comm
comp gen, complete, continue, declare, dirs, disown, echo, ena
eval, exec, exit, export, fc, fg, getopts, hash, help, histor
jobs, kill, let, local, logout, popd, printf, pushd, pwd, r
readonly, return, set, shift, shopt, source, suspend, test,
times, trap, type, typeset, ulimit, umask, unalias, unset, wa
```

本章将列出经常使用的内建命令并做简单的描述，在后面的章节中也会根据实际需要对部分命令做更详细的描述。

1.如何确定内建命令：type

不要试图用脑子记住所有的命令，这不可能也不需要。判断一个命令是不是内建命令只需要借助于命令type即可，如下所示：

```
#cd
命令是个内建命令
[root@localhost ~]# type cd
cd is a Shell builtin
#ifconfig
命令不是内建命令，而是一个外部文件
[root@localhost ~]# type ifconfig
ifconfig is /sbin/ifconfig
```

2.执行程序：“.”（点号）

点号用于执行某个脚本，甚至脚本没有可执行权限也可以运行。有时候在测试运行某个脚本时可能并不想为此修改脚本权限，这时候就可以使用“.”来运行脚本。以之前的HelloWorld.sh为例，如果没有运行权限的话，用“./”执行就会有报错，但是若在其前面使用点号来执行就不会报错，如下所示：

```
#
如果脚本没有可执行权限，则会报权限错误
[root@localhost ~]# ./HelloWorld.sh
-bash: ./HelloWorld.sh: Permission denied
#
使用点号执行没有加执行权限的脚本可以正常运行
[root@localhost ~]# . ./HelloWorld.sh
Hello World
```

与点号类似，`source`命令也可读取并在当前环境中执行脚本，同时还可返回脚本中最后一个命令的返回状态；如果没有返回值则返回0，代表执行成功；如果未找到指定的脚本则返回false。

```
[root@localhost ~]# source HelloWorld.sh
Hello World
```

3.别名：alias

`alias`可用于创建命令的别名，若直接输入该命令且不带任何参数，则列出当前用户使用了别名的命令。现在你应该能理解类似`ll`这样的命令为什么与`ls-l`的效果是一样的吧。

```
[root@localhost ~]# alias
alias cp='cp -i'
alias l.='ls -d .* --color=tty'
alias ll='ls -l --color=tty'
alias ls='ls --color=tty'
alias mv='mv -i'
alias rm='rm -i'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot --show-tilde'
```

使用`alias`可以自定义别名，比如说一般的关机命令是`shutdown-h now`，写起来比较长，这时可以重新定义一个关机

命令，以后就方便多了。使用alias定义的别名命令也是支持Tab键补全的，如下所示：

```
[root@localhost ~]# alias myShutdown='shutdown -h now'
```

注意，这样定义alias只能在当前Shell环境中有效，换句话说，重新登录后这个别名就消失了。为了确保永远生效，可以将该条目写到用户家目录中的.bashrc文件中，如下所示：

```
[root@localhost ~]# cat .bashrc
# .bashrc
# User specific aliases and functions
alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'
#
自定义关机命令的别名
myShutdown='shutdown -h now'
# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi
```

4.删除别名：unalias

该命令用于删除当前Shell环境中的别名。有两种使用方法，第一种用法是在命令后跟上某个命令的别名，用于删除指定的别名。第二种用法是在命令后接-a参数，删除当前Shell环境中所有的别名。同样，这两种方法都是在当前Shell环境中生效的。

```
#
删除11
别名
```

```
[root@e-bai ~]# unalias ll
#
再运行该命令时，报找不到该命令的错误。说明该别名被删除了
[root@e-bai ~]# ll
-bash: ll: command not found
```

5.任务前后台切换: bg、fg、jobs

该命令用于将任务放置后台运行，一般会与Ctrl+z、fg、&符号联合使用。典型的使用场景是运行比较耗时的任务。比如打包某个占用较大空间的目录，若在前台执行，在任务完成前将会一直占用当前的终端，而导致无法执行其他任务，此时就应该将这类任务放置后台。

```
[root@localhost ~]# tar -zcf usr.tgz /usr
tar: Removing leading '/' from member names  #
开始打包
#
占用前台导致无法运行其他任务，此处用Ctrl+z
组合键暂停前台任务
[1]+  Stopped                  tar -zcf usr.tgz /usr
[root@localhost ~]# jobs  #
查看暂停的任务，刚刚的tar
任务编号为1
[1]+  Stopped                  tar -zcf usr.tgz /usr
[root@localhost ~]# bg 1  #
把tar
任务放置后台继续运行
[1]+ tar -zcf usr.tgz /usr &    #tar
任务继续运行了
[root@localhost ~]# fg 1  #
使用fg
把后台任务调至前台运行
tar -zcf usr.tgz /usr
#
如果预知某个任务耗时很久，可以一开始就将命令放入后台运行
[root@localhost ~]# tar -zcf usr.tgz /usr &
```

6.改变目录: cd

改变当前工作目录。如果不加参数，默认会进入当前用户的家目录。

7.声明变量：declare、typeset

这两个命令都是用来声明变量的，作用完全相同。很多语法严谨的语言（比如C语言）对变量的声明都是有严格要求的：变量的使用原则是必须在使用前声明、声明时必须说明变量类型，而Shell脚本中对变量声明的要求并不高，因为Shell弱化了变量的类概念，所以Shell又被称为弱类型编程语言，声明变量时并不需要指明类型。不过，若使用declare命令，可以用-i参数声明整型变量，如下所示：

```
#
声明变量i_num01
，其值为1
[root@localhost ~]# i_num01=1
#
声明变量f_num01
，其值为3.14
[root@localhost ~]# f_num01=3.14
#
声明变量str01
，其值为HelloWorld
[root@localhost ~]# str01="HelloWorld"
#
使用declare
声明整型变量i_num02
，其值为1
[root@localhost ~]# declare -i i_num02=1
```

使用-r声明变量为只读，如下所示：

```
[root@localhost ~]# declare -r readonly=100 #
声明只读变量
[root@localhost ~]# readonly=200 #
试图改变变量值
```

-bash: readonly: readonly variable #
报错，提示尝试修改只读变量

使用-a声明变量，如下所示：

```
[root@localhost ~]# declare -a arr='([0]="a" [1]="b" [2]="c")'
[root@localhost ~]# echo ${arr[0]}
a
[root@localhost ~]# echo ${arr[1]}
b
[root@localhost ~]# echo ${arr[2]}
c
```

使用-F、-f显示脚本中定义的函数和函数体，如下所示：

```
#
创建脚本fun.sh
， 内容如下
[root@localhost ~]# cat fun.sh
#!/bin/bash
func_1()
{
    echo "Funciotn 1"
}
func_2()
{
    echo "Function 2"
}
echo "declare -F:"
declare -F
echo
echo "declare -f:"
declare -f
#
运行该脚本的输出效果如下
[root@localhost ~]# bash fun.sh
declare -F:
declare -f func_1
declare -f func_2
declare -f:
```

```
func_1 ()
{
    echo "Funciotn 1"
}
func_2 ()
{
    echo "Function 2"
}
```

8.打印字符：echo

echo用于打印字符，典型用法是使用echo命令并跟上使用双引号括起的内容（即需要打印的内容），该命令会打印出引号中的内容，并在最后默认加上换行符。使用-n参数可以不打印换行符。

```
[root@localhost ~]# echo "Hello World"
Hello World
[root@localhost ~]# #
命令提示符出现在新的一行
[root@localhost ~]# echo -n "Hello World"
Hello World[root@localhost ~]# #
命令提示符在同一行
```

默认情况下，echo命令会隐藏-e参数（禁止解释打印反斜杠转义的字符）。比如“\n”代表新的一行，如果尝试使用echo输出一行，在不加参数的情况下只会将“\n”当作普通的字符，若要打印转义字符，则需要通过使用-e参数来允许。

```
#echo
默认禁止打印反斜杠转义的字符
[root@localhost ~]# echo "\n"
\n #
只是把"\n"
"当做普通的字符
#
为了允许打印转义字符，需要使用 -e
```

参数

#

下面的输出有两行，第一行是输出的新行，第二行是默认的换行符

```
[root@localhost ~]# echo -e "\n"
```

```
[root@localhost ~]#
```

9.跳出循环：break

从一个循环（for、while、until或者select）中退出。break后可以跟一个数字n，代表跳出n层循环，n必须大于1，如果n比当前循环层数还要大，则跳出所有循环。下面的脚本演示了使用break和break 2的区别（运行前请将对应的注释符去掉）。

#

创建演示脚本break_01.sh

```
[root@localhost ~]# cat break_01.sh
```

```
#!/bin/bash
```

```
for I in A B C D
```

```
do
```

```
    echo -n "$I:"
```

```
    for J in `seq 10`
```

```
    do
```

```
        if [ $J -eq 5 ]; then
```

```
            break
```

```
            #break 2
```

```
        fi
```

```
        echo -n " $J"
```

```
    done
```

```
    echo
```

```
done
```

```
echo
```

#

判断当J

值为5

时，break

的输出结果（循环运行了4

次）

```
[root@localhost ~]# bash break_01.sh
```

```
A: 1 2 3 4
```

```
B: 1 2 3 4
```

```
C: 1 2 3 4
D: 1 2 3 4
#
判断当J
值为5
时, break 2
的输出结果 (仅运行了1
次循环便终止了)
[root@localhost ~]# bash break_01.sh
A: 1 2 3 4
```

10.循环控制: continue

停止当前循环, 并执行外层循环 (for、while、until或者select) 的下一循环。continue后可以跟上一个数字n, 代表跳至外部第n层循环。n必须大于1, 如果n比当前循环层数还要大, 将跳至最外层的循环。下面的脚本演示了使用continue和continue 2的区别 (运行前请将对应的注释符去掉)。

```
#
创建演示脚本continue_01.sh
[root@localhost ~]# cat continue_01.sh
#!/bin/bash
for I in A B C D
do
    echo -n "$I:"
    for J in `seq 10`
    do
        if [ $J -eq 5 ]; then
            continue
            #continue 2
        fi
        echo -n " $J"
    done
    echo
done
echo
#
判断当J
值为5
时, continue
```


的输出结果

```
[root@localhost ~]# bash continue_01.sh
```

```
A: 1 2 3 4 6 7 8 9 10
```

```
B: 1 2 3 4 6 7 8 9 10
```

```
C: 1 2 3 4 6 7 8 9 10
```

```
D: 1 2 3 4 6 7 8 9 10
```

```
#
```

判断当J

值为5

时, continue 2

的输出结果

```
[root@localhost ~]# bash continue_01.sh
```

```
A: 1 2 3 4B: 1 2 3 4C: 1 2 3 4D: 1 2 3 4
```

11.将所跟的参数作为Shell的输入，并执行产生的命令：eval

```
#eval
```

用法例一：将字符串解析成命令执行

```
#
```

定义cmd

为一个字符串，该字符串为“ls -l /etc/passwd”

```
"
```

```
[root@localhost ~]# cmd="ls -l /etc/passwd"
```

```
#
```

如果使用eval

，则会将之前的字符串解析为命令并执行

```
[root@localhost ~]# eval $cmd
```

```
-rw-r--r-- 1 root root 1638 Mar  3 00:43 /etc/passwd
```

```
#eval
```

用法例二：程序运行中根据某个变量确定实际的变量名

```
[root@localhost ~]# name1=john #
```

定义变量name1

```
[root@localhost ~]# name2=wang #
```

定义变量name2

```
[root@localhost ~]# num=1 #
```

使用该变量确定真实的变量名name\$num

```
[root@localhost ~]# eval echo "$"name$num
```

```
John
```

```
#eval
```

用法例三：将某个变量的值当做另一个变量名并给其赋值

```
[root@localhost ~]# name1=john
```

```
[root@localhost ~]# name2=wang
```

```
[root@localhost ~]# eval $name1="$name2" #  
等价于john="wang"  
[root@localhost ~]# echo $john  
wang
```

12. 执行命令来取代当前的Shell: exec

内建命令exec并不启动新的Shell，而是用要被执行的命令替换当前的Shell进程，并且将老进程的环境清理掉，而且exec命令后的其他命令将不再执行。假设在一个Shell里面执行了exec echo"Hello"命令，在正常地输出一个“Hello”后Shell会退出，因为这个Shell进程已被替换为仅仅执行echo命令的一个进程，执行结束自然也就退出了。如图11-5所示，命令执行完成后，连接状态是一个红色的断开符。

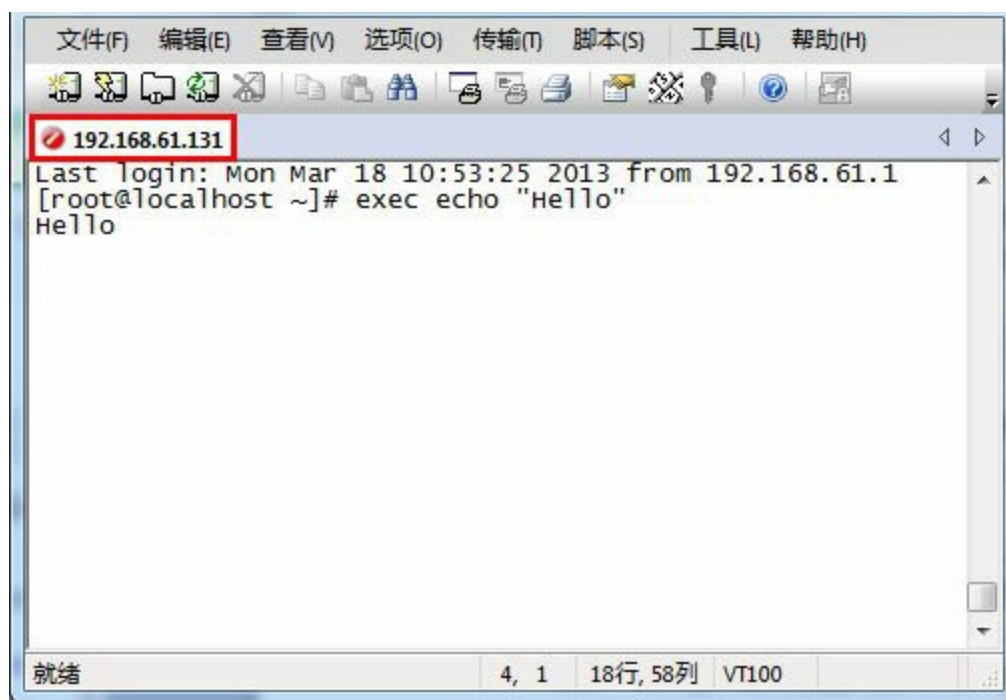


图11-5 exec执行后退出Shell

想要避免出现这种情况，一般将exec命令放到一个Shell脚本里面，由主脚本调用这个脚本，主脚本在调用子脚本执行时，当执行到exec后，该子脚本进程就被替换成相应的exec的

命令。注意source命令或者点号，不会为脚本新建Shell，而只是将脚本包含的命令在当前Shell执行。exec典型的用法是与find联合使用，用find找出符合匹配的文件，然后交给exec处理，如下所示：

```
#
列出系统中所有以.conf
结尾的文件
[root@localhost ~]# find / -name "*.conf" -exec ls -l {} \;
#
删除系统中所有临时文件
find / -name "*.tmp" -exec rm -f {} \;
```

13.退出Shell: exit

在当前Shell中直接运行该命令的后果是退出本次登录。在Shell脚本中使用exit代表退出当前脚本。该命令可以接受的参数是一个状态值n，代表退出的状态，下面的脚本什么都不会做，一旦运行就以状态值为5退出。如果不指定，默认状态值是0。

```
#
脚本exit.sh
的内容
[root@localhost ~]# cat exit.sh
#!/bin/bash
exit 5
#
[root@localhost ~]# bash exit.sh          #
运行该脚本
[root@localhost ~]#                      #
看起来什么都没有发生，实际上并不是这样
[root@localhost ~]# echo $?              #
使用$?
可以取出之前命令的退出状态值
5          #
这就是脚本中定义的退出状态值
```

14.使变量能被子Shell识别： export

用户登录到系统后，系统将启动一个Shell，用户可以在该Shell中声明变量，也可以创建并运行Shell脚本，通常，如果说登录时的Shell是父Shell，则在该Shell中运行的Shell是该Shell的子Shell。当子Shell运行完毕后，将返回执行该脚本的父Shell。从这种意义上来说，用户可以有许多Shell，每个Shell都是由父Shell创建的。

在父Shell中创建变量时，这些变量并不会被其子Shell进程所知，也就是说变量默认情况下是“私有”的，或称“局部变量”。使用export命令可以将变量导出，使得该Shell的子Shell都可以使用该变量，这个过程称为变量输出。

为演示export的作用，请先在当前Shell中创建文件export.sh，其内容如下所示：

```
#
创建export.sh
脚本
[root@localhost ~]# cat export.sh
#!/bin/bash
echo $var
#
直接执行这个脚本，由于变量var
在脚本中并没有定义，所以其值是空，脚本输出确实什么也没有
[root@localhost ~]# bash export.sh
[root@localhost ~] #
无任何输出
#
现在在当前Shell
中创建变量var
，并赋值为100
，并尝试输出该值
[root@localhost ~]# var=100
[root@localhost ~]# echo $var
100      #
确实变量var
```

被赋值了

由于这里的var
和子Shell
中的var
都是局部变量，所以如果现在再运行子Shell
，依然会打印出空值
[root@localhost ~]# bash export.sh
[root@localhost ~] #
无任何输出

但是如果在定义变量的时候使用了export
就不一样了，子Shell
可以读取该变量
[root@localhost ~]# export var=100
[root@localhost ~]# bash export.sh
100 #
这里读取到了父Shell
的变量var
值

要说明的是，即便子Shell确实读取到了父Shell中变量var的值，也只是值的传递，如果在子Shell中尝试改变var的值，改变的只是var在子Shell中的值，父Shell中的该值并不会因此受到影响，你可以认为父Shell和子Shell都各自拥有一个叫var的变量，它们恰巧名字相同而已。

15.发送信号给指定PID或进程：kill

Linux是一个多任务的操作系统，系统上经常同时运行着多个进程。我们需要知道如何控制这些进程。Linux操作系统包括3种不同类型的进程，第一种是交互进程，这是由一个Shell启动的进程，既可以在前台运行，也可以在后台运行；第二种是批处理进程，与终端没有联系，是一个进程序列；第三种是监控进程，也称系统守护进程，它们往往在系统启动时启动，并保持在后台运行。

kill命令用来终止进程，其工作的原理是向系统的内核发送

一个系统操作信号和某个程序的进程标识号，然后系统内核就可以对进程标识号指定的进程进行操作。比如用ps命令可以看到许多进程，有时需要使用kill中止某些进程来提高系统资源。该命令可以向某个PID或进程发送信号，具体用法在前面的第7章中已做详细的描述，此处不赘述。

```
[root@localhost ~]# kill [ -s signal | -p ] [ -a ] [ -  
- ] pid ...  
[root@localhost ~]# kill -l [ signal ]  
#-s:  
指定要发送的信号，信号可以是信号名或是信号数值  
#-p:  
只打印出进程的PID  
，并不真的发送信号  
#-l:  
指定信号的名称列表  
#pid:  
进程的ID  
号  
#signal:  
信号
```

16.整数运算：let

let是Shell内建的整数运算命令。以下是let的具体用法：

```
#let  
使用范例  
let I=2+2    --->I=4  
let J=5-2    --->J=3  
let K=2*5    --->K=10  
let L=15/7    --->L=2  
（整数计算，所以计算结果也是整数）  
let M=15%7    --->M=1  
（求余）  
let N=2**3    --->N=8  
（代表2  
的3  
次方）
```

```
#let
也支持类C
的计算方式
let i++
(i
自增1
)
let i--
(i
自减1
)
let i+=10
(i
值等于i
增加10
)
let i-=10
(i
值等于i
减少10
)
let i*=10
(i
值等于i
乘以10
)
let i/=10
(i
值等于i
除以10
)
let i%=10
(i
值等于i
模10
)
```

17.显示当前工作目录: pwd

pwd命令会打印当前工作目录的绝对路径名。如果使用-P选项，打印出的路径名中不会包含符号连接。如果使用了-L选项，打印出的路径中可以包含符号连接。如下所示：

```
#
运行pwd
可以显示当前所在目录的绝对路径
[root@localhost ~]# pwd
/root
#
变量OLDPWD
记录了上一次工作目录，如果你从登录系统之后一直没有改变工作目录，则
OLDPWD
为空
[root@localhost ~]# echo $OLDPWD
#
这里为空
#
变量PWD
记录了当前的工作目录，它与pwd
命令运行结果是一致的
[root@localhost ~]# echo $PWD
/root
#
进入/tmp
目录
[root@localhost ~]# cd /tmp
#
这时OLDPWD
记录了上一次工作目录/root
，所以此处输出为/root
[root@localhost ~]# echo $OLDPWD
/root
#
下面将演示-P
和-L
参数
#
首先确定/var/mail
目录其实是一个软连接
[root@localhost ~]# ls -l /var/mail
lrwxrwxrwx 1 root root 10 Nov 27 17:54 /var/mail -
> spool/mail
#
进入/var/mail
目录
[root@localhost ~]# cd /var/mail
#
使用-L
参数和不加参数的pwd
命令输出结果是一样的
```



```
[root@localhost mail]# pwd -L
/var/mail
#
使用 -P
参数则显示出真实的路径，而不是软链接
[root@localhost mail]# pwd -P
/var/spool/mail
```

18.声明局部变量：local

该命令用于在脚本中声明局部变量，典型的用法是用于函数体内，其作用域也在声明该变量的函数体中。如果试图在函数外使用local声明变量，则会提示错误。

19.从标准输入读取一行到变量：read

有时候我们开发的脚本必须具有交互性，也就是在运行过程中依赖人工输入才能继续。比如说，一箱啤酒有12瓶，买n箱啤酒一共多少瓶？由于此处n是一个变量，如果脚本能在运行中询问“你想买多少箱啤酒”，然后计算出一共有多少瓶啤酒的话，脚本会显得更为友好。

```
#
根据输入的箱数计算一共有多少瓶啤酒
[root@localhost ~]# cat read.sh
#!/bin/bash
declare N
echo "12 bottles of beer in a box"
echo -n "How many box do you want:"
read N
echo "$((N*12)) bottle in total"
#
运行效果
[root@localhost ~]# bash read.sh
12 bottles of beer in a box
How many box do you want:10 #
这里输入数字
120 bottle in total
```

从上面的例子中可以看到，我们通过read从键盘输入中读取到变量N的值使用了两句代码，实际上read可以使用-p参数代替。

```
#
原先用于读取变量N
使用的两句代码
echo -n "How many box do you want:"
read N
#read
使用 -p
参数简化后的代码
read -p "How many box do you want:" N
```

如果不指定变量，read命令会将读取到的值放入环境变量REPLY中。另外要记住，read是按行读取的，用回车符区分一行，你可以输入任意文字，它们都会保存在变量REPLY中。

```
[root@localhost ~]# read #
输入read
命令后回车
Hello world #
输入Hello world
[root@localhost ~]# echo $REPLY #
打印环境变量REPLY
Hello world #
结果与之前的输入一致
```

20. 定义函数返回值：return

典型的用于函数中，常见用法是return n，其中n是一个指定的数字，使函数以指定值退出。如果没有指定n值，则返回状态是函数体中执行的最后一个命令的退出状态。

```
[root@localhost ~]# cat return.sh
```

```
#!/bin/bash
#
# 定义了一个函数fun_01
# 该函数简单地返回1
function fun_01 {
    return 1
}
#
# 调用该函数
fun_01
#
# 查看之前函数的返回值
echo $?
#
# 实际运行一下这个脚本的效果
[root@localhost ~]# bash return.sh
1
```

21.向左移动位置参数: shift

要想搞清楚shift的用法，首先需要了解脚本“位置参数”的概念。假设一个脚本在运行时可以接受参数，那么从左到右第一个参数被记作\$1，第二个参数为\$2，以此类推，第n个参数为\$N。所有参数记作\$@或\$*，参数的总个数记作\$#，而脚本本身记作\$0。

```
#
# 通过阅读该脚本了解$0
# $1
# $2
# $3
# $@
# $#
# 等符号的含义
[root@localhost ~]# cat shift_01.sh
#!/bin/bash
echo "This script's name is:$0"
echo "The First parameter is:$1"
echo "The Second parameter is:$2"
echo "The Third parameter is:$3"
echo "All of the parameters are $@"
```

```
echo "Count of parameter is:$#"
#
运行该脚本的效果
[root@localhost ~]# bash shift_01.sh 1 2 3
This script's name is:shift_01.sh
The First parameter is:1
The Second parameter is:2
The Third parameter is:3
All of the parameters are 1 2 3
Count of parameter is:3
```

shift命令可以对脚本的参数做“偏移”操作。假设脚本有A、B、C这3个参数，那么\$1为A，\$2为B，\$3为C；shift一次后，\$1为B，\$2为C；再次shift后\$1为C，如下所示：

```
#
通过阅读该脚本了解shift
命令的作用
[root@localhost ~]# cat shift_02.sh
#!/bin/bash
until [ -z "$1" ]
do
    echo "$@"
    shift
done
#
运行该脚本，使用A B C
这3
个参数
[root@localhost ~]# bash shift_02.sh A B C
A B C
B C
C
```

22.显示并设置进程资源限度：ulimit

系统的可用资源是有限的，如果不限制用户和进程对系统资源的使用，则很容易陷入资源耗尽的地步，而使用ulimit可以控制进程对可用资源的访问。默认情况下Linux系统的各个

资源都做了软硬限制，其中硬限制的作用是控制软限制（换言之，软限制不能高于硬限制）。使用ulimit-a可以查看当前系统的软限制（使用命令ulimit-a-H可查看系统的硬限制）。下面是该命令的运行结果，笔者对每行输出都做了注解。

```
[root@localhost ~]# ulimit -a
#core
文件大小，单位是block
，默认为0
core file size          (blocks, -c) 0
#
数据段大小，单位是kbyte
，默认不做限制
data seg size           (kbytes, -d) unlimited
#
调度优先级，默认为0
scheduling priority      (-e) 0
#
创建文件的大小，单位是block
，默认不做限制
file size                (blocks, -f) unlimited
#
挂起的信号数量，默认是8192
pending signals          (-i) 8192
#
最大锁定内存的值，单位是kbyte
，默认是32
max locked memory        (kbytes, -l) 32
#
最大可用的常驻内存值，单位是kbyte
，默认不做限制
max memory size          (kbytes, -m) unlimited
#
最大打开的文件数，默认是1024
open files               (-n) 1024
#
管道最大缓冲区的值
pipe size                (512 bytes, -p) 8
#
消息队列的最大值，单位是byte
POSIX message queues     (bytes, -q) 819200
#
程序的实时性优先级，默认为0
real-time priority       (-r) 0
```

```
#
栈大小，单位是kbyte
stack size                (kbytes, -s) 10240
#
最大cpu
占用时间，默认不做限制
cpu time                  (seconds, -t) unlimited
#
用户最大进程数，默认是8192
max user processes        (-u) 8192
#
最大虚拟内存，单位是kbyte
，默认不做限制
virtual memory            (kbytes, -v) unlimited
#
文件锁，默认不做限制
file locks                (-x) unlimited
```

每一行中都包含了相应的改变该项设置的参数，以最大可以打开的文件数为例（`open files`默认是1024），想要增大至4096则按照如下命令设置（可参照此方法调整其他参数）。

```
#
设置最大打开的文件数
#
该命令会同时设置硬限制和软限制
[root@localhost ~]# ulimit -n 4096
#
使用-S
参数单独设置软限制
#[root@localhost ~]# ulimit -S -n 4096
#
使用-H
参数单独设置硬限制
#[root@localhost ~]# ulimit -H -n 4096
```

使用`ulimit`直接调整参数，只会在当前运行时生效，一旦系统重启，所有调整过的参数就会变回系统默认值。所以建议将所有的改动放在`ulimit`的系统配置文件中。相关配置方法请参考笔者对相关配置文件的注释。

```

[root@localhost ~]# cat /etc/security/limits.conf
# /etc/security/limits.conf
#
该文件是ulimit
的配置文件，任何对系统的ulimit
的修改都应该写入该文件
#
请将所有的设置写到该文件的最后
#Each line describes a limit for a user in the form:
#
配置应该写成下面这行的格式，即每个配置占用1
行，每行4
列
#
每列分别是<domain> <type> <item> <value>
#<domain>          <type>  <item>  <value>
#
#
其中：
#<domain>
可以取的值如下：
#          -
一个用户名
#          -
一个组名，组名前面用@
符号
#          -
通配符*
#          -
通配符%
#Where:
#<domain> can be:
#          - an user name
#          - a group name, with @group syntax
#          - the wildcard *, for default entry
#          - the wildcard %, can be also used with %group synta
#                  for maxlogin limit
#
#<type>
只有以下两个可用值：
#          - soft
用于设置软限制
#          - hard
用于设置硬限制
#<type> can have the two values:
#          - "soft" for enforcing the soft limits
#          - "hard" for enforcing hard limits

```

```

#
#<item>
的值可以是以下任意一种：
#           - core - core
文件大小的限制 (KB)
#           - data -
最大数据段限制 (KB)
#           - fsize -
最大文件大小 (KB)
#           - memlock -
最大锁定的内存大小 (KB)
#           - nofile -
最大打开文件数
#           - rss -
最大常驻内存值 (KB)
#           - stack -
最大栈空间大小 (KB)
#           - cpu -
最大CPU
使用时间 (MIN)
#           - nproc -
最大进程数
#           - as -
虚拟地址空间
#           - maxlogins -
某用户的最大登录数
#           - maxsyslogins -
系统用户最大登录数
#           - priority -
用户进程的运行优先级
#           - locks
-
用户最大可以锁定文件的数量
#           - sigpending -
最大挂起的信号量数
#           - msgqueue - POSIX
信号队列使用的最大内存值 (bytes)
#           - nice -
最大nice
值
#           - rtprio -
最大实时优先级
#
#<item> can be one of the following:
#           - core - limits the core file size (KB)
#           - data - max data size (KB)
#           - fsize - maximum filesize (KB)
#           - memlock - max locked-in-memory address space (KB)

```



```

#         - nofile - max number of open files
#         - rss - max resident set size (KB)
#         - stack - max stack size (KB)
#         - cpu - max CPU time (MIN)
#         - nproc - max number of processes
#         - as - address space limit
#         - maxlogins - max number of logins for this user
#         - maxsyslogins - max number of logins on the system
#         - priority - the priority to run user process with
#         - locks - max number of file locks the user can hold
#         - sigpending - max number of pending signals
#         - msgqueue - max memory used by POSIX message queues
#         - nice - max nice priority allowed to raise to
#         - rtprio - max realtime priority
#
#<domain>      <type>  <item>          <value>
#
#
以下是使用样例，请参照配置
#*              soft    core              0
#*              hard    rss                10000
#@student       hard    nproc              20
#@faculty       soft    nproc              20
#@faculty       hard    nproc              50
#ftp            hard    nproc              0
#@student       -       maxlogins          4

```

23.测试表达式: test

该命令用于测试表达式EXPRESSION的值，根据测试结果返回0（测试失败）或1（测试成功）。由于该命令非常强大且在Shell脚本中非常重要，所以将会专门使用一节来讲解。其基本用法如下：

```
[root@localhost ~]# test EXPRESSION
```

第12章 Bash Shell的安装

在第11章中我们系统了解了Bash Shell的发展历史以及Bash Shell中常见的内建命令，一般而言，Bash Shell是很多Linux发行版的默认Shell，所以会随着系统的安装而自动安装。不过确实有一部分读者想要安装较新版本的Bash Shell，所以本章会具体讲一下其安装方法，希望可以作为读者全新安装Bash Shell或者虽然已经安装但希望升级的参考。

12.1 确定你的Shell版本

如果你安装的Linux是RedHat、CentOS、Fedora、Ubuntu、Debian等主流发行版，那么在你的系统中很可能已经预装了Bash Shell，只需要确认一下是否确实已经安装以及预装的版本即可。具体的方法是：

```
#
确认系统中使用的Shell
是bash
[root@localhost ~]# echo $SHELL
/bin/bash
#
查看系统中Bash Shell
的版本（方法一）
[root@localhost ~]# bash --version
GNU bash, version 3.2.25(1)-release (i686-redhat-linux-gnu)
Copyright (C) 2005 Free Software Foundation, Inc.
#
查看系统中Bash Shell
的版本（方法二）
[root@localhost ~]# echo $BASH_VERSION
3.2.25(1)-release
```

12.2 安装bash

正如第8章所说，Linux下安装软件的方式无非是RPM包安装、yum安装、源码安装3种方式，读者可以任选一种方式。不过，相对来说RPM包安装和yum安装方式比较简单，若再考虑各种包的依赖关系，这两种方式中又属yum安装更为简单。因而这里就不详细介绍这两种安装方法了，下面会具体示范使用源码安装bash的过程。

首先访问<http://www.gnu.org/software/bash/bash.html>页面，在Downloads中选择一个下载的连接，笔者选择了中国科技大学提供的FTP下载目录：<ftp://mirrors.ustc.edu.cn/gnu/bash/>。

当前很多生产环境的系统中使用的bash版本还是3.2版，读者可以根据实际需要选择具体的版本。在笔者撰写本书时，最新的版本是4.2版本，所以这里使用这个版本来做示范。

第一步：使用wget下载最新的bash源码包，如下所示：

```
[root@localhost ~]# wget ftp://mirrors.ustc.edu.cn/gnu/bash/b
4.2.tar.gz
--2013-04-11 19:37:41-
- ftp://mirrors.ustc.edu.cn/gnu/bash/bash-4.2.tar.gz
=> `bash-4.2.tar.gz'
Resolving mirrors.ustc.edu.cn... 202.141.160.110, 2001:da8:d8
Connecting to mirrors.ustc.edu.cn|202.141.160.110|:21... conn
Logging in as anonymous ... Logged in!
==> SYST ... done. ==> PWD ... done.
==> TYPE I ... done. ==> CWD /gnu/bash ... done.
==> SIZE bash-4.2.tar.gz ... 7009201
==> PASV ... done. ==> RETR bash-4.2.tar.gz ... done.
Length: 7009201 (6.7M)
100%
[=====>] 7,009,201 1.9
2013-04-11 19:37:46 (1.89 MB/s) - `bash-
4.2.tar.gz' saved [7009201]
```

第二步：解压源码包，并进入生成的目录中。

```
#
解压后会在当前目录下生成一个bash-4.2
目录
[root@localhost ~]# tar zxvf bash-4.2.tar.gz
#
进入目录bash-4.2
[root@localhost ~]# cd bash-4.2
[root@localhost bash-4.2]#
```

第三步：准备配置（configure）。

最简单的配置方式是直接运行当前目录下的configure，这会将bash安装到/usr/local目录中，不过编译安装软件时，好的习惯是使用--prefix参数指定安装目录。所以这里采用下面的配置方式。该条命令将会产生大量的输出，一开始会检查系统的编译环境以及相关的依赖软件。最常见的错误可能是系统中没有安装gcc造成无法继续（见下面输出内容中画横线的部分），如果是这个原因，使用yum install gcc命令或者参照8.2.4节的安装方式进行安装。如果配置过程出现致命错误会立即退出，请读者注意输出内容中的error部分。

```
[root@localhost bash-4.2]# ./configure --
prefix=/usr/local/bash4.2
checking build system type... i686-pc-linux-gnu
checking host system type... i686-pc-linux-gnu
Beginning configuration for bash-4.2-release for i686-pc-
linux-gnu
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... Yes
.....(
略去内容).....
#
如果大量的checking
没问题，则配置环境检测通过。如果读者看到如下的输出内容，说明配置成功
```

```
configure: creating ./config.status
config.status: creating Makefile
config.status: creating builtins/Makefile
config.status: creating lib/readline/Makefile
config.status: creating lib/glob/Makefile
config.status: creating lib/intl/Makefile
config.status: creating lib/malloc/Makefile
config.status: creating lib/sh/Makefile
config.status: creating lib/termcap/Makefile
config.status: creating lib/tilde/Makefile
config.status: creating doc/Makefile
config.status: creating support/Makefile
config.status: creating po/Makefile.in
config.status: creating examples/loadables/Makefile
config.status: creating examples/loadables/perl/Makefile
config.status: creating config.h
config.status: executing default-1 commands
config.status: creating po/POTFILES
config.status: creating po/Makefile
config.status: executing default commands
#
如果配置成功，会在当前目录中生成Makefile
[root@localhost bash-4.2]# ll Makefile
-rw-r--r-- 1 root root 77119 Apr 11 19:49 Makefile
```

第四步：正式编译。

```
#
编译过程会产生大量输出
[root@localhost bash-4.2]# make
rm -f mksyntax
gcc -DPROGRAM='"bash"' -DCONF_HOSTTYPE='"i686"'
-DCONF_OSTYPE='"linux-gnu"' -DCONF_MACHTYPE='"i686-pc-linux-
gnu"'
-DCONF_VENDOR='"pc"'
-DLOCALEDIR='"/usr/local/bash4.2/share/locale"'
-DPACKAGE='"bash"' -DSHELL -DHAVE_CONFIG_H -I. -I. -
I./include
-I./lib -g -o mksyntax ./mksyntax.c
.....(
略去内容).....
```

第五步：安装。有时在安装前也可以进行测试，但是一般

情况下这不是必需的。

```
#
非必要步骤：测试安装
#[root@localhost bash-4.2]# make test
#
安装
[root@localhost bash-4.2]# make install
#
安装其实就是将make
产生的文件复制到指定的目录中，在这里指定的目录就是之前我们用
--prefix
参数指定的/usr/local
，可以在该目录中发现bash4.2
目录
[root@localhost ~]# ls -ld /usr/local/bash4.2/
drwxr-xr-x 4 root root 4096 Apr 11 20:08 /usr/local/bash4.2/
```

到此为止，最新版本的bash就已经安装好了，确切地说是安装到了/usr/local/bash4.2中。

12.3 使用新版本的Bash Shell

虽然最新版的bash已经安装到系统中，但是还需要经过一些设置才能使用。首先需要将最新的bash的路径写到/etc/shells中，以向系统注册新Shell的路径。可以采取直接编辑/etc/shells文件的方式，或者采用如下更简单的方式：

```
[root@localhost ~]# echo "/usr/local/bash4.2/bin/bash" >> /et
```

然后使用命令chsh（change shell的简写）修改登录Shell。

```
[root@localhost ~]# chsh
Changing shell for root.
New shell [/bin/bash]: /usr/local/bash4.2/bin/bash #
输入要修改的shell
Shell changed. #
显示成功修改了shell
#
此处chsh
并没有附加参数，所以默认是修改root
的shell
，如要改变其他用户的登录shell
，
可以在后面跟上用户名，使用这种方式给用户john
更改shell
[root@localhost ~]# chsh john
```

chsh命令做的工作就是修改了/etc/passwd文件中登录Shell的路径，所以如果明白了chsh的原理，实际上可以手工编辑/etc/passwd文件，将root用户的这行改成下面的样子（这又一次印证了Linux中一切皆文件的说法）：

```
[root@localhost ~]# cat /etc/passwd | grep bash4.2
```



```
root:x:0:0:root:/root:/usr/local/bash4.2/bin/bash
```

最后还需要重新登录以获得Shell，登录后再次验证一下当前的Shell版本。

```
[root@localhost ~]# echo $BASH_VERSION
4.2.0(1)-release
#
请注意，如果这时候你使用下面的命令可能会犯迷糊：为什么版本是3.2.25
呢？不是已经是4.2
了吗？
[root@localhost ~]# bash --version
GNU bash, version 3.2.25(1)-release (i686-redhat-linux-gnu)
Copyright (C) 2005 Free Software Foundation, Inc.
#
通过使用whereis bash
命令可了解当前运行的bash
命令真实运行的是/bin/bash
，也就是说
现在是在版本为4.2
的bash
中运行了一个3.2.25
版本的bash
命令。如果要想每次运行bash
的
时候使用的是4.2
的版本，需要修改PATH
变量的值，读者可以自行完成这个任务
[root@localhost ~]# whereis bash
bash: /bin/bash /usr/local/bash4.2 /usr/share/man/man1/bash.1
```

12.4 在Windows中安装bash

Linux是学习Bash Shell的天然环境，但是借助工具，在Windows下同样可以运行bash。最著名的工具是Cygwin，它是模拟类UNIX环境的软件，最初由Cygwin Solution公司开发，目的在于通过重新编译将Linux系统上的软件移植到Windows上。

安装Cygwin需要到其官网<http://www.cygwin.com/>上下载安装包。在该网站首页的Current Cygwin DLL version中，找到setup.exe并下载。该安装程序只是一个“壳子”，或者说它更应该被称为安装Cygwin的安装器，因为该文件只有不到1MB的大小。双击该程序进入安装界面，然后单击“下一步”按钮，如图12-1所示。

在随后的页面中，保持Install from Internet选中，然后单击“下一步”按钮，如图12-2所示。

安装目录使用默认的C:\cygwin，然后单击“下一步”按钮，如图12-3所示。



图12-1 安装Cygwin的界面

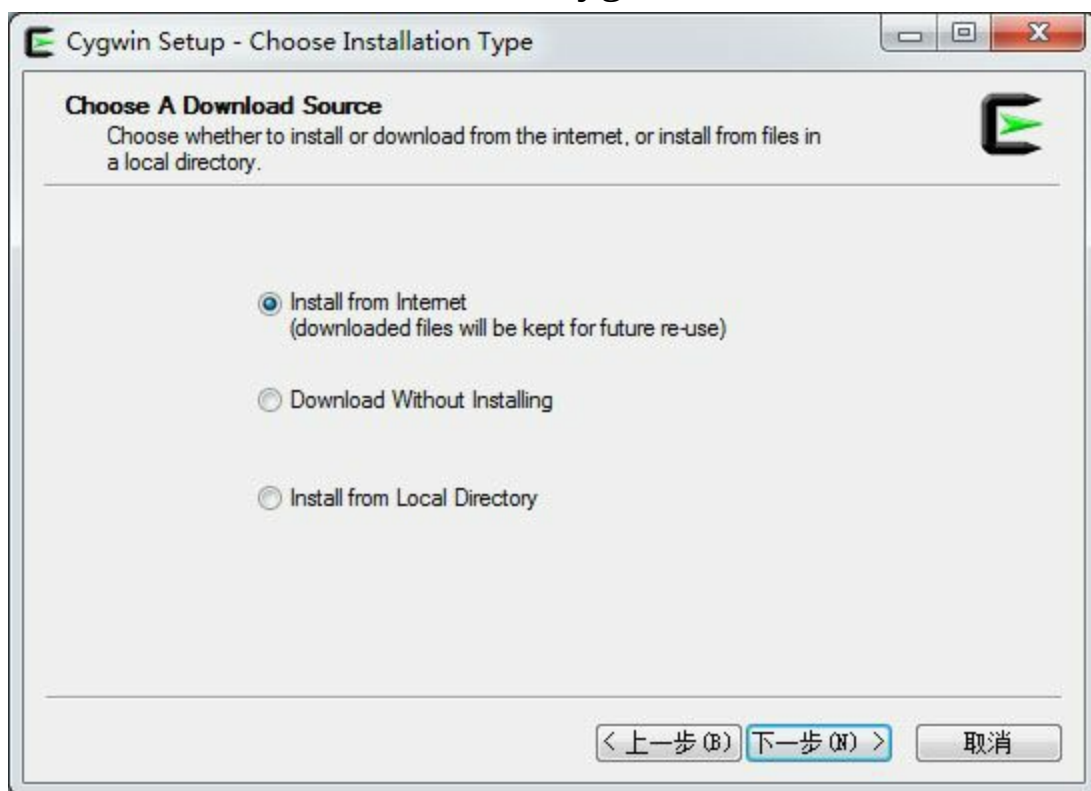


图12-2 从网络下载并安装Cygwin及其组件

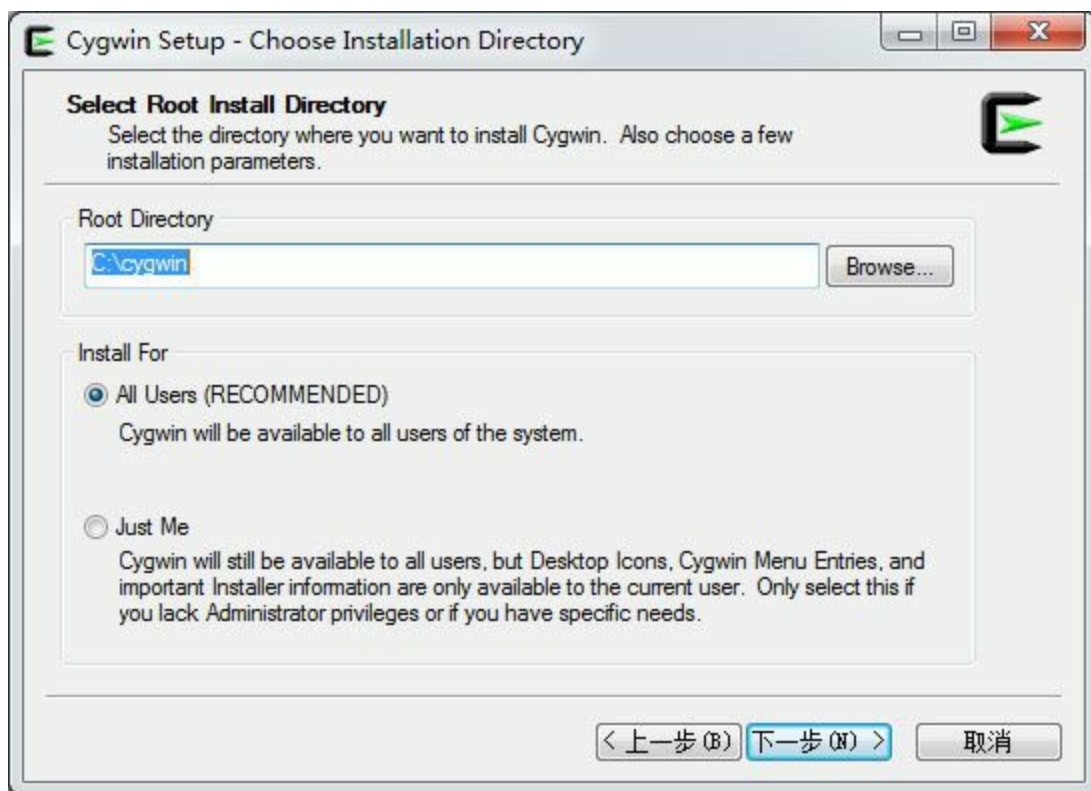


图12-3 选择安装目录

此时需要指定一个本地目录用于存放下载的安装文件，如果指定的目录不存在则会自动创建该目录，这里保持系统默认目录不变，如图12-4所示。

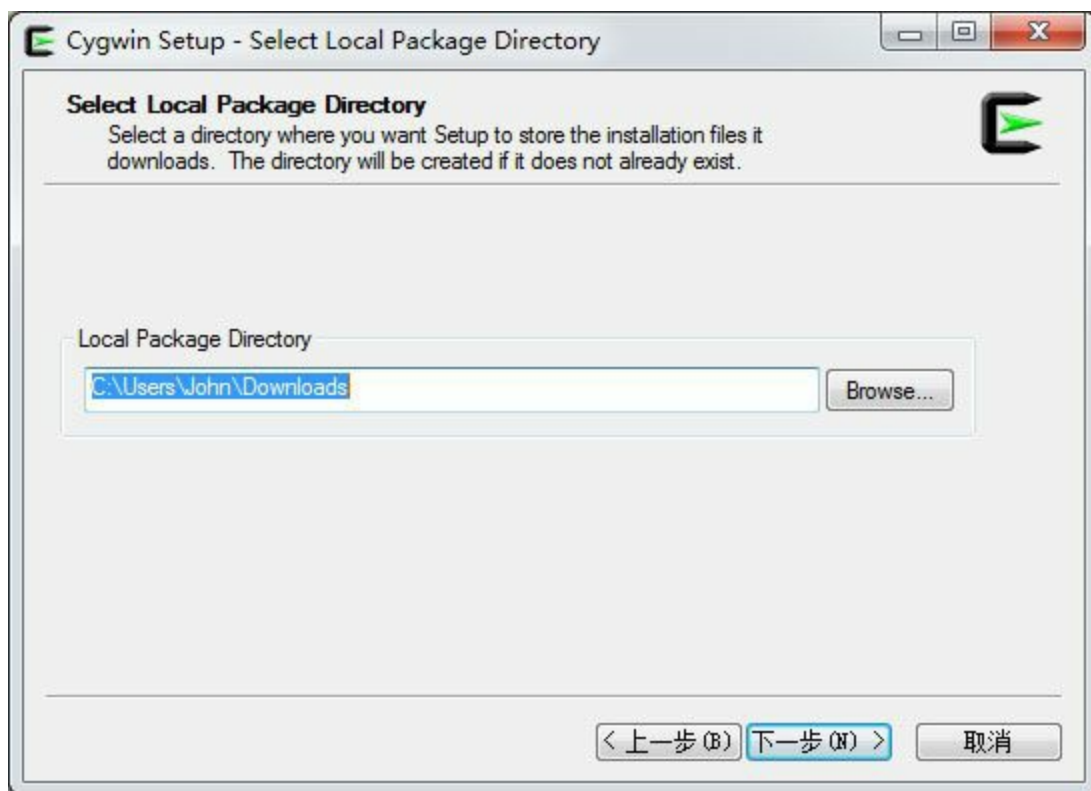


图12-4 指定存放下载文件的目录

指定下载使用的网络连接方式，如果读者的网络可以直接访问外部网络，则选取默认的Direct Connection即可；如果不能直接上网，请咨询网络管理人员，或通过网络代理服务器下载安装，如图12-5所示。

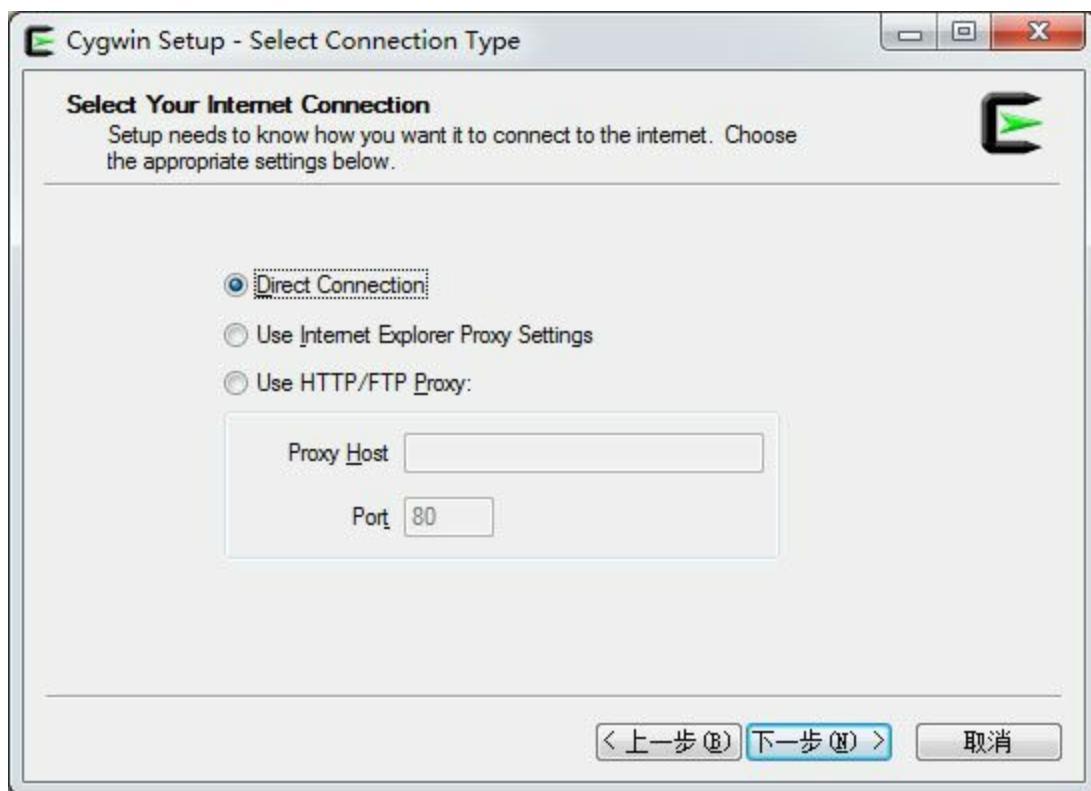


图12-5 选择网络连接方式

选择下载源，国内可以选用163的站点，速度相对不错，如图12-6所示。

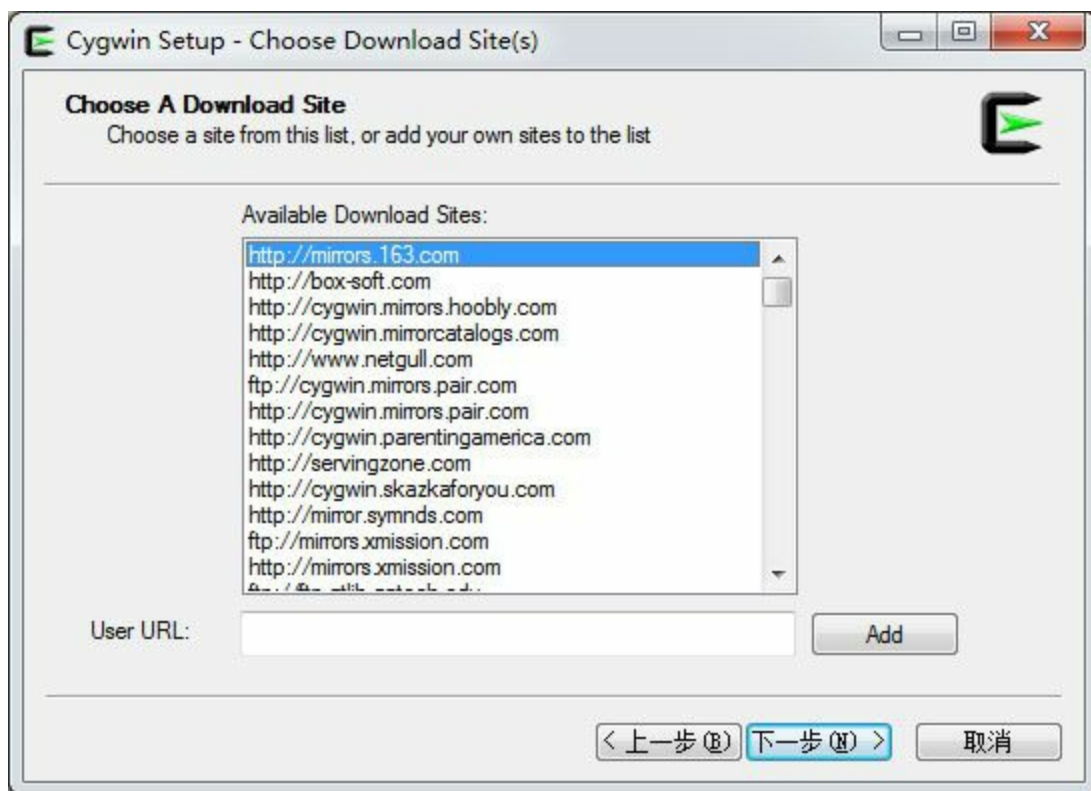


图12-6 选择下载源

搜索bash，并在出现的Shells Default中选择bash的版本，然后单击“下一步”按钮，如图12-7所示。安装完成界面如图12-8所示。

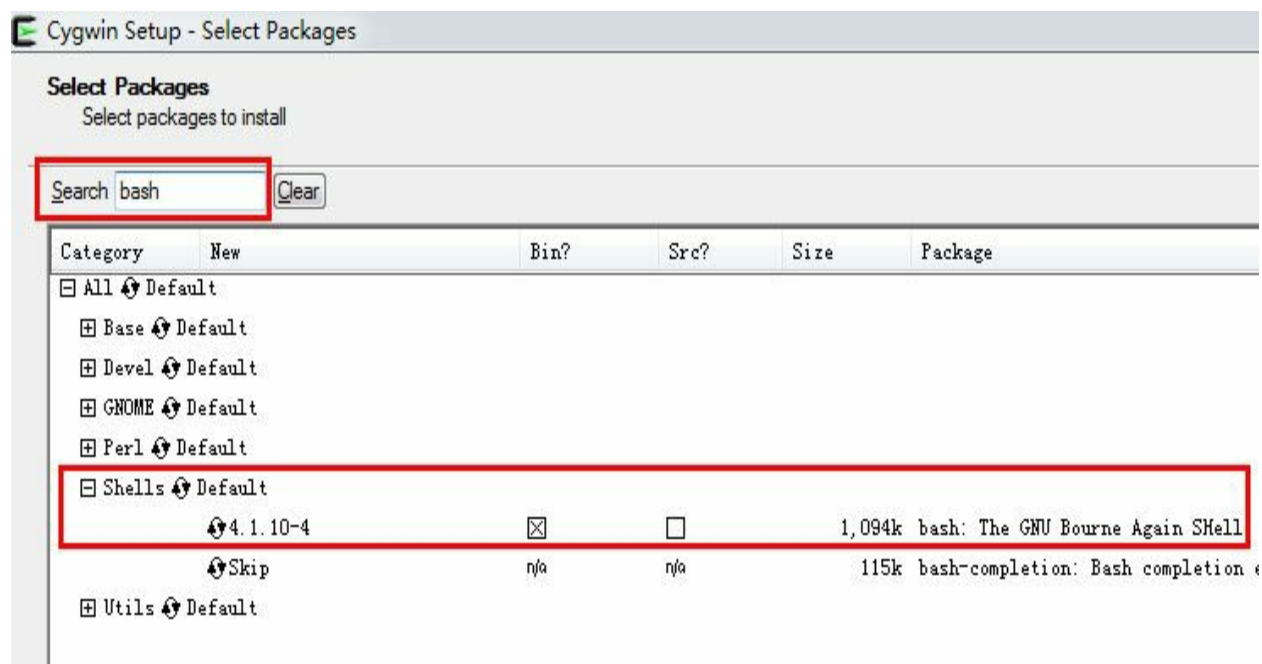


图12-7 选择bash的版本

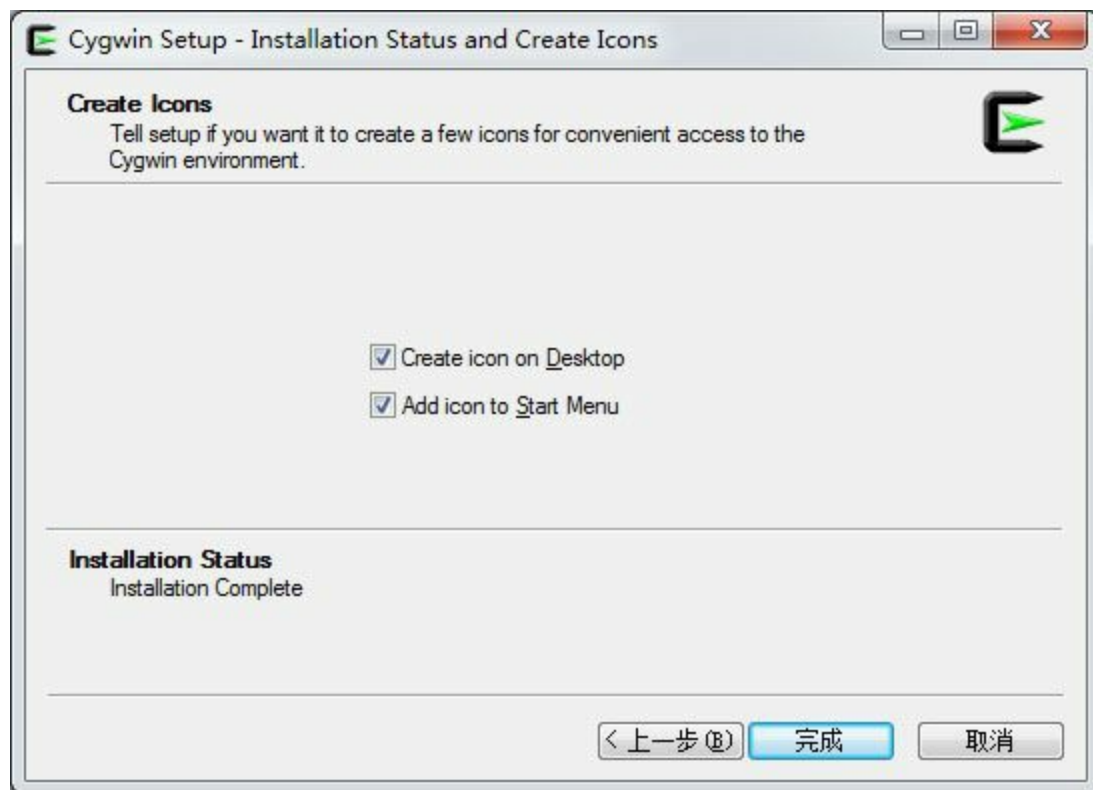
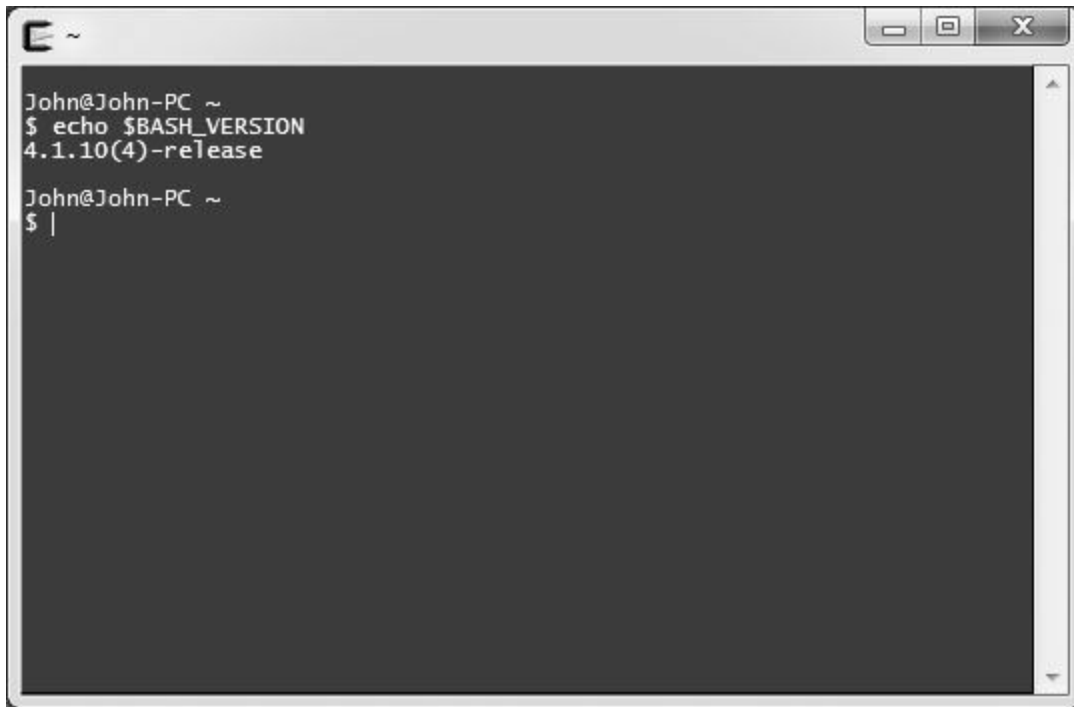


图12-8 安装完成

最后，在桌面上找到Cygwin Terminal，启动后运行命令查看当前bash的版本，如图12-9所示。至此在Windows平台下的

bash环境就安装完了，读者可以在该窗口下尝试运行一些bash命令。



```
John@John-PC ~  
$ echo $BASH_VERSION  
4.1.10(4)-release  
John@John-PC ~  
$ |
```

图12-9 启动Cygwin Terminal

第13章 Shell编程基础

13.1 变量

顾名思义，变量就是其值可以变化的量。从变量的实质来说，变量名是指向一片用于存储数据的内存空间。变量有局部变量、环境变量之分。Shell变量是一种弱类型的变量，也就是说在声明变量时并不需要指定其变量类型，而且也不需要遵循C语言中“先声明再使用”的规定，任何时候只要想用就可以用。在脚本中，往往需要使用变量来存储有用信息，比如文件名、路径名、数值等，通过这些变量可以控制脚本的运行行为。

13.1.1 局部变量

所谓局部变量就是指在某个Shell中生效的变量，对其他Shell来说无效，而且会随着当前Shell的消失而消失，局部变量的作用域被限定在声明它们的Shell中，可以使用local内建命令来“显式”的声明局部变量，但仅限于函数内使用。换言之，每个Shell都有自己的变量空间，彼此互不影响。而环境变量不仅仅是对于该Shell生效，对其子Shell也同样生效。

想要进一步了解局部变量在不同Shell中的互不相关性，可在图形界面下同时打开两个Shell，或使用两个终端远程连接到服务器（SSH），在其中的一个Shell命令行中定义一个变量I并赋值为1，然后打印，这时在同一个Shell窗口中是可正确打印变量I的值的；与此同时，新打开一个Shell窗口，同样打印变量I的值，但结果却为空，如图13-1所示。这说明局部变量仅在定义该变量的Shell中生效，而对其他Shell没有影响。这很好理解，就像小王家和小徐家都有一部电视机（变量名相同），但是同一时刻小王家和小徐家的电视中播放的节目可以是不一样的（变量值不同）。

13.1.2 环境变量

环境变量通常又称“全局变量”，以区别于局部变量。在Shell脚本中，变量默认就是全局的，无论在脚本的任何位置声明，但是为了让子Shell继承当前Shell的变量，则可以使用`export`内建命令将其导出为环境变量。该命令的使用方法如下：

```
[root@localhost ~]# export VAR=value
```

其中，`VAR`是变量的名字，`value`为值，使用等号相连，注意等号两端没有空格。



图13-1 局部变量作用域演示

环境变量可用在创建变量的Shell和从该Shell派生的任意子Shell或进程中（使用`export`内建命令将变量导出为环境变量），因此，环境变量通常又被称作全局变量。环境变量被创建时所处的Shell被称为父Shell，如果在父Shell中再创建一个Shell，则该Shell被称作子Shell。当子Shell产生时，它会继承父Shell的环境变量为自己所用，所以说环境变量可从父Shell传给予Shell。不难理解，环境变量还可以传递给孙Shell。请注

意，环境变量只能向下传递而不能向上传递，即“传子不传父”。在一个Shell中创建子Shell最简单的方式是运行bash命令，如图13-2所示。

```
[root@localhost ~]#  
[root@localhost ~]# bash 运行 bash 命令并回车  
[root@localhost ~]# █ ———> 这里看起来好像没什么区别，其实不然  
                                这里已经进入了子Shell
```

图13-2 创建子Shell

为了演示环境变量对子Shell的影响，这里安排一个小实验。请读者使用SSH远程连接到一台服务器，输入用户名、密码后，将得到一个登录Shell，为方便起见，将其命名为父Shell；在该Shell中输入命令bash并回车，这时候进入子Shell；在子Shell中，声明一个环境变量export VAR=1，并确认VAR已经正确赋值（echo VAR）；然后在子Shell中再输入命令bash并回车，这时候进入孙Shell（也就是子Shell的子Shell，捋顺一下思绪，千万别乱了）。现在在孙Shell中使用echo VAR命令，发现可以打印出正确的值，这说明子Shell中的环境变量确实可以传递给孙Shell。最后，连续输入两次exit命令依次退出孙Shell和子Shell，进入父Shell中，此时再使用echo VAR命令，发现父Shell并没有这个值，这说明子Shell的环境变量并不能传递给父Shell，如图13-3所示。

bash中默认包含有几十个预设的环境变量，这里挑选常见的一些予以介绍。读者此处可以迅速浏览一遍，留个印象，今后在使用的过程中再慢慢熟悉。可以通过echo的方式查看这些变量的值。

```
[root@localhost ~]# bash
[root@localhost ~]#
[root@localhost ~]# export VAR=1
[root@localhost ~]# echo $VAR
1
[root@localhost ~]#
[root@localhost ~]# bash
[root@localhost ~]#
[root@localhost ~]# echo $VAR 孙Shell
1
[root@localhost ~]#
[root@localhost ~]# exit 子Shell
exit
[root@localhost ~]#
[root@localhost ~]# exit
exit
[root@localhost ~]#
[root@localhost ~]# echo $VAR 父Shell
1
[root@localhost ~]#
```

图13-3 环境变量的继承关系

变量：BASH

说明：Bash Shell的全路径。

```
[root@localhost ~]# echo $BASH
/bin/bash
```

变量：BASH_VERSION

说明：Bash Shell的版本。

```
[root@localhost ~]# echo $BASH_VERSION
3.2.25(1)-release
```

变量：CDPATH

说明：用于快速进入某个目录。在Linux管理中我们可能经常要进入网络配置的目录（/etc/sysconfig/network-scripts/）中

修改配置文件，但由于目录名较长，每次进入该目录都显得非常困难，如下所示：

```
[root@localhost ~]# cd /etc/sysconfig/network-scripts/
```

使用CDPATH变量，可以迅速地进入该目录，如下所示：

```
[root@localhost ~]# CDPATH="/etc/sysconfig/"  
#  
注意这里执行cd network-scripts  
时，先会在当前目录中查找是否有network-scripts  
,  
如果有，将会进入当前目录中的network-scripts  
目录；如果没有，才会进入CDPATH  
定义的  
目录中的network-scripts  
目录  
[root@localhost ~]# cd network-scripts  
/etc/sysconfig/network-scripts
```

变量：EUID

说明：记录当前用户的UID。当前的用户是root，所以该值应该为0。

```
[root@localhost ~]# echo $EUID  
0
```

变量：FUNCNAME

说明：在用户函数体内部，记录当前函数体的函数名。创建funcname.sh文件，内容如下（运行该脚本后注意查看脚本的输出）：

```
[root@localhost ~]# cat funcname.sh
#!/bin/bash
funcname() {
    echo $FUNCNAME
}
funcname
[root@localhost ~]# bash funcname.sh
funcname
```

变量：HISTCMD

说明：记录下一条命令在history命令中的编号。如果运行history命令查看到history中一共记录了1016条已经执行过的命令，则下面一条命令的编号将是1018（因为本次history为第1017条命令）。

```
[root@localhost ~]# history
.....(
略去内容).....
1016 history
[root@localhost ~]# echo $HISTCMD
1018
```

变量：HISTFILE

说明：记录history命令记录文件的位置。运行history命令将打印出已经运行过的命令列表，即便重启机器之后还能保存以前的命令记录。但这是如何做到的呢？其实history只不过是找到\$HISTFILE所指定的命令记录文件，并将其打印出来。一般默认每个用户的家目录下都有一个.bash_history文件，用于记录该用户运行过的命令历史记录。

```
[root@localhost ~]# echo $HISTFILE
```

/root/.bash_history

变量：HISTFILESIZE

说明：设置HISTFILE文件记录命令的行数。

如果任凭HISTFILE文件不断增大，显然会有一天这个文件将大到不可收拾，而且也没有必要记录那么多命令，所以使用某种机制限制该文件的大小是非常必要的，HISTFILESIZE就起着这样的作用。

```
[root@localhost ~]# echo $HISTFILESIZE
1000
```

变量：HISTSIZE

说明：事实上Linux并不会每次运行一个命令后立即将该命令记录到HISTFILE文件中，读者可以试着在当前的Shell中多运行几次命令，然后用cat/root/.bash_history命令查看。原因是Shell采用了“命令缓冲区”来记录所有已运行过的命令，在缓冲区满或退出Shell时才将缓冲区的记录写到HISTFILE文件中。缓冲区的大小使用HISTSIZE定义。

```
[root@localhost ~]# echo $HISTSIZE
1000
```

变量：HOSTNAME

说明：展示主机名。这个很简单，直接看代码，如下所示：

```
[root@localhost ~]# echo $HOSTNAME  
localhost.localdomain
```

变量：HOSTTYPE

说明：展示主机的架构，是i386、i686，还是x86_64等。
代码如下：

```
[root@localhost ~]# echo $HOSTTYPE  
i686
```

变量：MACHTYPE

说明：主机类型的GNU标识，这种标识有统一的结构。一般来说是“主机架构-公司-系统-gnu”，在RedHat系统中打印该变量值，如下所示：

```
[root@localhost ~]# echo $MACHTYPE  
i686-redhat-linux-gnu
```

变量：LANG

说明：设置当前系统的语言环境，其实就是language的意思。

```
#  
显示当前语言环境  
[root@localhost ~]# echo $LANG  
en_US.UTF-8  
#  
设置语言环境为中文  
[root@localhost ~]# export LANG=zh_CN.UTF-8
```

变量：PWD

说明：记录当前目录。

```
[root@localhost ~]# echo $PWD
/root
```

变量：OLDPWD

说明：记录之前目录，这个值是什么由之前所在的那个目录决定。

```
#
进入/mnt
目录
[root@localhost ~]# cd /mnt
#
进入/root
目录
[root@localhost mnt]# cd
#
看看之前在哪个目录
[root@localhost ~]# echo $OLDPWD
/mnt
```

变量：PATH

说明：这个变量在本书中多次出现了，代表命令的搜索路径，非常重要。前面的章节中已经很详细地描述了它的含义和用法，读者一定要理解并掌握。

```
#
查看当前PATH
```

的值
[root@localhost ~]# echo \$PATH
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/root/bin

重设PATH
变量，增加/some/path
至原来的PATH
中
[root@localhost ~]# export PATH=/some/path:\$PATH

变量：PS1

说明：命令提示符，默认值是[\u@\h\W]\\$，其中\u是用户名、\h是主机名、\W是当前工作目录的basename、\\$是用户UID的替换字符：如果UID是0则替换成“#”，否则替换成“\$”，所以此处具体显示出来就是“[root@localhost~]#”。该变量可以有很多种组合，可以根据自己的喜好进行定制。但是毕竟它没有太大的实际用途，所以不需要读者浪费太多时间学习。表13-1列举了一些可用的符号，供读者参考。

表13-1 一些命令提示符

标识符代码	含 义
\W	当前工作目录的 basename
\w	当前工作目录的全路径
\d	日期，格式为“周 月 日”
\H	完整的主机名
\h	主机名
\t	24 小时制的时间，格式为 HH: MM: SS
\T	12 小时制的时间
\u	当前用户的用户名
\\$	如果 UID 是 0 则显示“#” 否则显示“\$”

Shell 预设的变量还有很多，读者可以使用 `man bash` 查看 `man` 文件，在 `Shell Variables` 章节中可具体查看每个变量的含义。表13-2中简要描述了之前未讲到的预设变量及其用途。

表13-2 一些预设变量

变量名	用 途
BASH_ENV	一般该值为空。如果该变量在调用脚本时已经设置，它的值将被展开，并用作在执行脚本前读取的启动文件名
BASH_VERSINFO	一个只读变量数组，保存 bash 的版本信息
COLUMNS	决定 select 内建命令打印选择列表时的宽度
COMP_LINE	当前命令行
COMP_POINT	相对于当前命令起点的当前光标位置
COMP_WORDS	由当前命令行中单个词组成的变量数组
DIRSTACK	保存当前目录栈内容的变量数组
FIGNORE	由冒号分隔的在补全文件名时要忽略的后缀列表
GLOBIGNORE	由冒号分隔的模板列表，定义在文件名展开时忽略的文件名
GROUPS	一个数组变量，包含当前用户作为成员组的列表
HISTCONTROL	定义一个命令是否加入历史列表中
IGNOREEOF	控制 Shell 接收 EOF 字符作为独立输入的行为
INPUTRC	readline 初始化文件的名称，取代默认值 /etc/inputrc

LC_ALL	如果该变量设置了，则这个变量将覆盖 LANG 的值
LC_CTYPE	决定在文件名展开和模板匹配里字符的解释和字符集的行为
LC_MESSAGES	该变量决定用于转换由 \$ 引导的双引号字符串的区域
LC_NUMERIC	该变量决定数字格式化的本地类别
LINENO	当前执行的脚本或者 Shell 函数的行数
LINES	决定内建命令 select 打印选择列表的列长度
MAILCHECK	Shell 从 MAILPATH 或 MAIL 变量指定的文件中检查邮件的频率
OPTERR	如果设置成 1，bash 显示内建命令 getopts 生成的错误信息

(续)

变量名	用 途
PIPESTATUS	最近运行过的前台管道进程的退出状态值的列表
PPID	Shell 父进程的进程 ID
PS3	这个变量的值被用作 select 命令的提示符
PS4	在命令行前打印的提示符
RANDOM	生成一个 0~32767 的随机整数
REPLY	内建命令 read 的默认值
SECONDS	Shell 运行的秒数
SHELLOPTS	由冒号分隔的 Shell 已经启用的选项列表
SHLVL	每新增一个 Shell 进程，该值就增加 1
TMOUT	作为内建命令 read 的默认超时时间。当 Shell 处于交互状态时，这个值表示等待在基本提示串后输入的秒数
UID	当前用户的真实用户 ID

13.1.3 变量命名

Shell中的变量必须以字母或者下划线开头，后面可以跟数字、字母和下划线，变量长度没有限制。下面列举了一些变量命名，注意Shell的变量是区分大小写的，这也就表示firstname和FIRSTNAME是不同的两个变量。

```
#
正确的变量命名
firstname
FIRSTNAME
_helloworld
big_data
Fullname
Person01
#
错误的变量命名
51play #
变量不能以数字开头
*badname #
变量不能以特殊字符开头
PS1 #
变量不能和Shell
的预设变量名重名
for #
变量不能使用Shell
的关键字
```

按照以上的变量命名规则定义变量abc，从理论上来说是可行的，但是一个好的习惯是变量最好能表明它代表的含义。比如说变量Student_ID，一看就知道它所表达的是“学号”的意思，绝对比number这种模棱两可的变量更清晰，不仅看代码的人觉得简单明了，而且有利于后期的代码维护。更好的习惯则是加上一些注释，但也不要太过拘泥，如下所示：

```
#
```

```
定义学号      #  
使用注释解释变量使后期阅读更为清晰  
Student_ID  
#  
定义一个日期  #  
这种注释就显得有所拘泥  
DATE
```

13.1.4 变量赋值和取值

变量赋值的方法是非常简单明了的，如下所示：

```
#
定义变量：变量名=
变量值
#
注意点一：变量名和变量值之间用等号紧紧相连，之间没有任何空格
[root@localhost ~]# name=john #
定义变量
#
如果不注意，等号任何一边出现空格就会出错
[root@localhost ~]# name = john
-bash: name: command not found
[root@localhost ~]# name="john" #
这样也是可以的
#
注意点二：当变量中有空格时必须用引号括起，否则会出现错误
#
其中的引号可以是双引号，也可以是单引号
[root@localhost ~]# name="john wang"
#[root@localhost ~]# name=john wang
#-bash: wang: command not found
```

变量的取值也很简单，只需要在变量名前加上\$符号既可，严谨一点的写法是\${}，如下所示：

```
[root@localhost ~]# echo $name
john wang
[root@localhost ~]# echo ${name}
john wang
#
使用${}
获取变量值是一种相对比较保险的方式
[root@localhost ~]# name="sue "
#
这里本是想打印名字，后面跟Hello
的，但Shell
```

试图将nameHello

解释为一个变量。

从Shell

语法来说，也确实应该将nameHello

解释为变量

```
[root@localhost ~]# echo $nameHello
#
```

因为变量nameHello

未声明，所以值为空

```
[root@localhost ~]# echo ${name}Hello
sue Hello
#
```

注意点三：如果变量值引用的是其他变量，则必须使用双引号。因为单引号会阻止Shell

解释特殊字符\$

```
[root@localhost ~]# name=john
[root@localhost ~]# name1="$name"
[root@localhost ~]# echo $name1
john
[root@localhost ~]# name1='$name'
[root@localhost ~]# echo $name1
$name
```

由于Shell具有“弱变量”的特性，因此即便在没有预先声明变量的时候也是可以引用的，而且没有任何报错或者提醒，这可能会造成脚本中引用不正确的变量，从而导致脚本异常，但是却很难找出原因。在这种情况下，可以设置脚本运行时必须遵循“先声明再使用”的原则，这样一旦脚本中出现使用未声明的变量的情况则立刻报错，如下所示：

```
[root@localhost ~]# echo $unDefinedVar
#
```

因为该变量未声明，所以值为空，但没有任何报错

#

设置变量必须先声明再使用

```
[root@localhost ~]# shopt -s -o nounset
[root@localhost ~]# echo $unDefinedVar
-bash: unDefinedVar: unbound variable
```

13.1.5 取消变量

取消变量指的是将变量从内存中释放，使用的命令是 `unset`，后面跟变量名。函数也是可以被取消的，所以 `unset` 后面还可以跟上函数名以取消函数。命令如下：

```
#
取消变量
[root@localhost ~]# name=john
[root@localhost ~]# echo $name
john
此时变量有值
[root@localhost ~]# unset name
[root@localhost ~]# echo $name
#
变量中已经没有内容了
[root@localhost ~]#
#
取消函数
unset_function() {
    echo "Hello World"
}
unset unset_function
unset_function #
由于函数已被取消，这里调用会出错
```

13.1.6 特殊变量

1.位置参数

Shell中还有一些预先定义的特殊只读变量，它们的值只有在脚本运行时才能确定。首先是“位置参数”，位置参数的命名简单直接，比如，脚本本身为\$0，第一个参数为\$1，第二个参数为\$2，第三个为\$3，以此类推。当位置参数的个数大于9时，需要用\${}括起来标识，比如说第10个位置参数应该记为\${10}。另外，\$#表示脚本参数的个数总和，\$@或\$*表示脚本的所有参数。下面的示例脚本使用了这些特殊的位置参数，请注意不同位置参数的输出。

```
[root@localhost ~]# cat posion.sh
#!/bin/bash
echo "This script's name is: $0"
echo "$# parameters in total"
echo "All parameters list as: $@"
echo "The first parameter is $1"
echo "The second parameter is $2"
echo "The third parameter is $3"
[root@localhost ~]# bash posion.sh para1 para2 para3
This script's name is: posion.sh
3 parameters in total
All parameters list as: para1 para2 para3
The first parameter is para1
The second parameter is para2
The third parameter is para3
```

2.脚本或命令返回值：\$?

在管理员登录到系统中交互式地输入命令时，系统也会及时在屏幕上输出内容给予反馈。比如说本想使用ifconfig查看网卡状态，但是将命令错写成ifconfi，系统会立刻给出command not found的提示，这种提示确实能让管理员感觉到系

统非常“友好”。

```
[root@localhost ~]# ifconfi
-bash: ifconfi: command not found
```

但是有很多后台脚本是需要每天自动运行的，比如每天凌晨两点的数据库备份。在这种情况下一旦出错是不可能第一时间知道的。那靠什么判断出错了呢？

再考虑一个情景：有些自动备份脚本在按时完成本地数据备份后，还会复制一份放在远程主机上（通过scp就可以做到）。不过在复制前需要先确认远程主机是否“活着”，这可以通过ping远程主机做到。如果能ping通则进行复制，如果ping不通则采取其他动作。这里又如何判断是否ping成功了呢？

这时就需要借助命令的返回值来判断了。Linux中规定正常退出的命令和脚本应该以0作为其返回值，任何非0的返回值都表示命令未正确退出或未正常执行。

在第一个例子中，输错命令后立即查看当时特殊变量\$?的值为127；第二个例子中，ping不通某个地址时查看当时的\$?值为1。注意，\$?永远是上一个命令的返回值，所以要查看某个命令的返回值必须在运行该命令后立即查看\$?。在自动化脚本中，也可以通过\$?变量的值判断之前命令的执行状态，从而采取不同的动作。

```
#
输入错误的命令时的返回值
[root@localhost ~]# ifconfi
-bash: ifconfi: command not found
[root@localhost ~]# echo $?
127
#
尝试ping
```


主机ping

不通时的返回值

```
[root@localhost ~]# ping 192.168.61.100 -c 1
PING 192.168.61.100 (192.168.61.100) 56(84) bytes of data.
From 192.168.61.131 icmp_seq=1 Destination Host Unreachable
--- 192.168.61.100 ping statistics ---
1 packets transmitted, 0 received, +1 errors, 100% packet loss
[root@localhost ~]# echo $?
```

1

13.1.7 数组

数组是一种特殊的数据结构，其中的每一项被称为一个元素，对于每个元素，都可以用索引方式取出元素的值。使用数组的典型场景是一次性要记录很多类型相同的数据时（但不是说一定要相同，因为Shell变量是弱类型性的，并不要求数组的每个元素都是相同类型）。比如，为了记录班级中所有人的数学成绩，如果不用数组来处理那就只能定义所有人成绩的变量，这就会显得非常烦琐。Shell中的数组对元素个数没有限制，但只支持一维数组，这一点和很多语言不同。

1.数组定义

数组的定义方法如下：用declare命令定义数组Array，并将第一个元素赋值为0，第二个元素赋值为1，其下标（也就是索引）则分别是0和1（记住数组的索引是从0开始计数的），然后打印出第一个元素的值。

```
#  
定义名为Array  
的索引数组  
[root@localhost ~]# declare -a Array  
#  
数组的下标从0  
开始计数，定义了第一个元素值为0  
，第二个元素值为1  
[root@localhost ~]# Array[0]=0  
[root@localhost ~]# Array[1]=1
```

如果说数组Array的前两个元素“类型相同”（严格意义上这么说是不对的），那么第三个元素就显得“另类”了：赋值为一个字符串。这又一次验证了Shell变量是弱类型的，这在很多语言中是不可能的。

```
[root@localhost ~]# Array[2]="HelloWorld"
```

和其他变量一样，Shell中对于数组变量的声明也非常宽松，而且随时都可以根据需要增加变量中的元素。相比其他语言，Shell的数据更为灵活。在很多语言中，一旦对数组进行了初始化就不能再改变大小了。

```
#  
数组还可以在创建的同时赋值  
[root@localhost ~]# declare -a Name=('john' 'sue')  
#  
增加元素  
[root@localhost ~]# Name[2]='wang'  
#  
更简单的创建数组的方式——不使用declare  
关键字  
#[root@localhost ~]# Name=('john' 'sue')
```

还可以给特定的元素赋值。下面的示例就是只对第四个、第六个、第八个元素进行赋值。

```
#  
跳号赋值  
[root@localhost ~]# Score=([3]=3 [5]=5 [7]=7)
```

2.数组操作

1) 数组取值：知道了如何定义数组和赋值元素后，下面就要了解数组的一些常见操作。最简单的操作就是数组取值，其格式为：**`${数组名[索引]}`**。以之前定义的数组Array、Name为例，取值演示如下：

```
#
取数组第一个元素的值
[root@localhost ~]# echo ${Array[0]}
0
[root@localhost ~]# echo ${Array[2]}
HelloWorld
#
打印数组中的元素值
[root@localhost ~]# echo ${Name[0]}
john
[root@localhost ~]# echo ${Name[1]}
sue
```

指定索引只能列举单个元素，要是想一次性取出所有元素的值，可以采取以下两种方式：

```
[root@localhost ~]# echo ${Array[@]}
0 1 HelloWorld
[root@localhost ~]# echo ${Array[*]}
0 1 HelloWorld
```

从表面上来看两者没有什么区别，但是`${Array[@]}`得到的是以空格隔开的元素值，而`${Array[*]}`的输出是一整个字符串。

2) 数组长度：即数组元素个数。利用“@”或“*”字符，可以将数组扩展成列表，然后使用“#”来获取数组元素的个数，如下所示：

```
[root@localhost ~]# echo ${#Array[@]}
3
[root@localhost ~]# echo ${#Array[*]}
3
```

如果某个元素是字符串，还可以通过指定索引的方式获得该元素的长度，如下所示：

```
[root@localhost ~]# echo ${#Array[2]}  
10
```

3) 数组截取：可以截取某个元素的一部分，对象可以是整个数组或某个元素。

```
#  
取出数组的第一、第二个元素  
[root@localhost ~]# echo ${Array[@]:1:2}  
1 HelloWorld  
#  
取出第二个元素从第0  
个字符开始连续5  
个字符  
[root@localhost ~]# echo ${Array[2]:0:5}  
Hello
```

4) 连接数组：将若干个数组进行拼接操作。

```
[root@localhost ~]# Conn=(${Array[@]} ${Name[@]})  
[root@localhost ~]# echo ${Conn[@]}  
0 1 HelloWorld john sue
```

5) 替换元素：将数组内某个元素的值替换成其他值。

```
[root@localhost ~]# Array=(${Array[@]/HelloWorld/HelloJohn})  
[root@localhost ~]# echo ${Array[@]}  
0 1 HelloJohn
```

6) 取消数组或元素：和取消一般变量一样，取消一个数组的方式也使用unset命令。

```
#
取消数组中的一个元素
[root@localhost ~]# unset Array[1]
[root@localhost ~]# echo ${Array[@]}
0 HelloJohn
#
取消整个数组
[root@localhost ~]# unset Array
[root@localhost ~]# echo ${Array[@]}
#
本行打印为空，说明Array
已经被取消了
[root@localhost ~]#
```

13.1.8 只读变量

只读变量又称常量，是通过readonly内建命令创建的变量。这种变量在声明时就要求赋值，并且之后无法修改，这和之前讲到的使用declare-r声明只读变量的效果是一致的。

```
#  
声明只读变量R0  
并赋值为100  
[root@localhost ~]# readonly R0=100  
#  
此处尝试修改R0  
的值，Shell  
会报错  
[root@localhost ~]# R0=200  
-bash: R0: readonly variable
```

13.1.9 变量的作用域

变量的作用域又叫“命名空间”，表示变量（identifier，标识符）的上下文。相同的变量可以在多个命名空间中定义，并且彼此之间互不干涉，所以在一个新的命名空间中可以自定义任何变量，因为所定义的变量都只在各自的命名空间中。就像A班有个小明，B班也有个小明一样，虽然他们都叫小明（对应于变量名），但是由于所在的班级不一样（对应于命名空间），所以这不会造成混乱。但是如果同一个班级中有两个小明，就必须用类似于“大小明”、“小小明”这样的命名来区分他们。

在Linux系统中，不同进程ID的Shell默认为一个不同的命名空间。下面的例子展示了同名变量在两个不同命名空间中是互不影响的。

```
#Namespace01.sh
中声明了VAR_01=100
[root@localhost ~]# cat Namespace01.sh
#!/bin/bash
VAR_01=100
echo VAR_01 in $0:$VAR_01
#
此处调用Namespace02.sh
bash Namespace02.sh
#Namespace02.sh
中声明了VAR_01=200
[root@localhost ~]# cat Namespace02.sh
#!/bin/bash
VAR_01=200
echo VAR_01 in $0:$VAR_01
#
执行Namespace01.sh
，看到同名变量VAR_01
在不同的脚本中的值是不一样的
[root@localhost ~]# bash Namespace01.sh
VAR_01 in Namespace01.sh:100
```


VAR_01 in Namespace02.sh:200

Shell变量的作用域是在本Shell内，属于本Shell的全局变量，也就是从定义该变量的地方开始到Shell结束，或到主动使用unset删除了该变量的地方为止。在变量的作用域内，该变量都是可见的，在函数内对变量也是可以访问、可修改的，这和C语言极为不同。

```
[root@localhost ~]# cat Namespace03.sh
#!/bin/bash
VAR_02=100
#
定义函数ch_var()
，该函数中重新定义VAR_02
并赋值为200
function ch_var() {
    VAR_02=200
}
echo "Before function VAR_02:$VAR_02"
#
调用函数
ch_var
echo "After function VAR_02:$VAR_02"
#Namespace03.sh
执行结果
[root@localhost ~]# bash Namespace03.sh
Brfore function VAR_02:100
After function VAR_02:200
```

同样的代码用C实现后，VAR_02的值并没有受到函数内部同名变量的影响。

```
[root@localhost ~]# cat Namespace03.c
#include <stdio.h>
int VAR_02=100;
void ch_var(void) {
    int VAR_02=200;
}
```

```
int main() {
    printf("Before function VAR_02: %d\n", VAR_02);
    ch_var();
    printf("After function VAR_02: %d\n", VAR_02);
}
#
执行结果
[root@localhost ~]# ./a.out
Before function VAR_02: 100
After function VAR_02: 100
```

存在这种差别的原因在于，Shell默认以Shell的进程ID作为一个命名空间，所以即便是在函数中声明变量，该变量也会在全局生效。而C语言会对函数内的变量单独创建命名空间，这样就不会影响全局定义的同名变量。

Shell的这种特性在一般情况下是没有太大问题的，但有时确实可能会给程序的开发造成麻烦，特别是当脚本实现了模块化的开发后，不同的人共同维护同一个脚本中不同功能的代码时，很可能大家都会用到比较常见的类似于i、j、k这样的临时变量（特别是在函数内部，使用这样的变量尤为常见），这无疑会造成问题。为了解决这种问题，在函数内部声明的临时变量需要用local指定其为只在函数内生效的“局部变量”，这样这些变量将只存在于局部的命名空间内，从而不会对全局变量有影响。下面按照这种方式对Namespace03.sh进行修改，在函数内部使用local声明变量VAR_02，再次执行然后查看效果。

```
[root@localhost ~]# cat Namespace03.sh
#!/bin/bash
VAR_02=100
function ch_var() {
    local VAR_02=200          #
    此处使用local
    声明变量
}
echo "Before function VAR_02:$VAR_02"
ch_var
echo "After function VAR_02:$VAR_02"
```

```
#  
修改后的Namespace03.sh  
执行结果  
[root@localhost ~]# bash Namespace03.sh  
Before function VAR_02:100  
After function VAR_02:100
```

从执行结果可以看到，在函数体内使用`local`关键字声明了和全局变量同名的局部变量后，对该变量的操作只会影响局部变量，而不会影响与之同名的全局变量。

13.2 转义和引用

Shell中有两类字符，一类是普通字符，在Shell中除了本身的字面意思外没有其他特殊意义，即普通纯文本（literal）；另一类即元字符（meta），是Shell的保留字符，在Shell中有着特殊意义。这在很多时候会造成麻烦：比如说想要在程序中用美元符打印商品的价格，但是这个符号一般被解析成提取变量的值。为了消除这些特殊字符的功能，就必须对其进行转义和引用。

13.2.1 转义

转义是指使用转义符引用单个字符，从而使其表达单纯的字符的字面含义。Shell中的转义符是反斜线“\”，使用转义符的目的是使转义符后面的字符单纯地作为字符出现，而不解释其特殊的含义——这是笔者尝试的最能表达转义符含义的解释，不过我相信如果不借助于实例还是非常难以理解的。下面就来看看实例。

请考虑如何使用echo打印出“\$Dollar”这种字符串。如果不假思索，你可能给出与下面类似的错误写法：

```
#
试图打印"$Dollar
"字符串的错误演示
[root@localhost ~]# echo $Dollar
---
```

此处打印为空，因为Shell
尝试打印出变量Dollar
的值，但是这个变量并没有声明，所以打印空行

```
#
使用转义字符转义$
字符
[root@localhost ~]# echo \$Dollar
$Dollar
```


更多的例子

```
#
打印乘号。如果不用转义符转义*
号，则*
号会作为一般的通配符使用，结果是将工作目录中的
所有目录和文件名替换它
[root@localhost ~]# echo 8 \* 8 =64
8 * 8 =64
```


句子中含有引号。如果不用转义符转义'
单引号，则Shell
会等待出现另一个单引号才能结束echo
进程

```
[root@localhost ~]# echo john\'s cat  
john's cat
```

在上面的示例中，命令的输出将会是空字符。由于“\$”符号是一个特殊字符，所以“\$Dollar”被Shell尝试解释为“取出并打印Dollar变量的值”，如果恰巧你在系统中定义了这个变量并赋予赋值，那么此处会打印出该变量的值——无论是哪种，都不是你原先想要的结果。这时就要使用“\”来转义“\$”字符，让“\$”失去其特殊含义，而只作为一个符号出现。

表13-3列出了在Shell中有特殊含义的字符，想要显示其字符本身，则需要使用转义符进行转义。不需要记住它们，但是建议多看几遍留个印象，在需要使用的时候查阅一下即可。

表13-3 Shell特殊字符

特殊字符	转义写法
' (单引号)	\'
" (双引号)	\"
* (星号)	*
%	\%
?	\?
\	\\
~	\~
` (反引号)	\`
+	\+
!	\!
#	\#
\$	\\$
&	\&
(\(
)	\)
[\[

]	\]
{	\{
}	\}
<	\<
>	\>
	\
;	\;
/	\

13.2.2 引用

引用是指将字符串用某种符号括起来，以防止特殊字符被解析为其他意思。比如说上一小节中的转义符就是一种引用。Shell中一共有4种引用符，分别是双引号、单引号、反引号（在键盘上和波浪号位于同一个键）和转义符。其中双引号又叫“部分引用”或“弱引用”，可以引用除\$符、反引号、转义符之外的所有字符；单引号又叫“全引用”或“强引用”，可以引用所有字符；反引号则会将反引号括起的内容解释为系统命令。

1.部分引用

部分引用是指用双引号括起来的引用。在这种引用方式中，\$符、反引号（`）、转义符（\）这3种特殊字符依然会被解析为特殊意义。比如，在定义一个变量后，使用echo打印该变量的时候，将它们用双引号括起来，如下所示：

```
#
声明变量VAR03
，并用echo
打印出来。第一次直接打印，第二次用引号括起来，从输出内容看好像没什么区别
[root@localhost ~]# VAR03=100
[root@localhost ~]# echo $VAR03
100
[root@localhost ~]# echo "$VAR03"
100
#
声明变量VAR03
，内容为字符串，ABC
之间有多个空格
[root@localhost ~]# VAR04="A   B   C"
#
直接打印变量时，输出内容只保留了每个字母间一个空格
[root@localhost ~]# echo $VAR04
A B C
#
使用引号括起的输出内容和变量定义时的内容是完全一致的
```

```
[root@localhost ~]# echo "$VAR04"  
A   B   C
```

2.全引用

全引用是指用单引号括起来的引用。单引号中的任何字符都只当作是普通字符（除了单引号本身，也就是说单引号中间无法再包含单引号，即使用转义符转义单引号也不行）。所有在单引号中的字符都只能代表其作为字符的字面意义。如果用单引号引用之前声明的变量，输出的内容如下所示：

```
[root@localhost ~]# echo '$VAR03'  
$VAR03  
[root@localhost ~]# echo '$VAR04'  
$VAR04
```

可以看到，输出内容就是单引号所括起来的所有内容，而不会将变量解析为其值。

如果全引用括起的字符串中还含有单引号，则会出现问题，因为Shell无法区分哪个单引号是引用的结束符，就像下面显示的一样：

```
[root@localhost ~]# echo 'It's a dog'  
>
```

想要解决这个问题，可以采取如下两种方式：

```
[root@localhost ~]# echo 'It'\''s a dog'  
It's a dog  
[root@localhost ~]# echo "It's a dog"  
It's a dog
```

单引号和双引号在很多时候是一样的，只是要记住，在双引号中的\$符、反引号、转义符还是会被解析成其特殊含义，而在单引号中所有的字符都只是字面意思。下面的例子中，使用双引号括起的内容中，\$PWD被解析成/root，而在单引号中只是按照原样输出“\$PWD”字符。

```
[root@localhost ~]# echo "Current directory is $PWD"
Current directory is /root
[root@localhost ~]# echo 'Current directory is $PWD'
Current directory is $PWD
```

13.2.3 命令替换

命令替换是指将命令的标准输出作为值赋给某个变量。比如，在某个目录中输入ls命令可查看当前目录中所有的文件，但如何将输出存入某个变量中呢？这就需要使用命令替换了，这也是Shell编程中使用非常频繁的功能。

Shell中有两种方式可以完成命令替换，一种是反引号（```），一种是`$()`，使用方法如下：

```
[root@localhost ~]# `
命令`
```

或

```
[root@localhost ~]# $(
命令)
```

运行系统命令date可以得到当前的系统时间。在很多时候我们需要记录脚本运行时间，所以只是运行这个命令是没有意义的，必须将该命令的运行结果记录并保存到变量中，并持久化到文件中，才能为后期分析提供有用的参考依据。

```
#
用两种命令替换方式记录date
命令的输出
[root@localhost ~]# DATE_01=`date`
[root@localhost ~]# DATE_02=$(date)
[root@localhost ~]# echo $DATE_01
Tue Apr 23 14:33:49 CST 2013
[root@localhost ~]# echo $DATE_02
Tue Apr 23 14:34:05 CST 2013
```

如果被引用的命令输出的内容包括多行，此时若不通过引用的方式输出变量，则输出的内容中将会删除换行符，文件名之间会使用系统默认的空来填充，即输出的内容只占一行。

```
[root@localhost ~]# LS=`ls -l`  
#  
不引用变量值的输出  
[root@localhost ~]# echo $LS  
total 36 -rw----- 1 root root 1017 Jan 2 2009 anaconda-  
ks.cfg  
-rw-r--r-- 1 root root 18590 Jan 2 2009 install.log  
-rw-r--r-- 1 root root 0 Jan 2 2009 install.log.syslog  
#  
引用变量值的输出  
[root@localhost ~]# echo "$LS"  
total 36  
-rw----- 1 root root 1017 Jan 2 2009 anaconda-ks.cfg  
-rw-r--r-- 1 root root 18590 Jan 2 2009 install.log  
-rw-r--r-- 1 root root 0 Jan 2 2009 install.log.syslog
```

以上使用反引号的部分都可以使用`$()`进行替换，因为它们都是等价的。但反引号毕竟和单引号看起来类似，有时候会对查看代码造成困难，而使用`$()`就相对清晰，能有效地避免这种混乱。但是有些情景是必须使用`$()`的：`$()`支持嵌套，而反引号不行。下面的例子演示了使用计算`ls`命令列出的第一个文件的行数，这里使用了两层嵌套。

```
[root@localhost ~]# Fir_File_Lines=$(wc -l $(ls | sed -  
n '1p'))  
[root@localhost ~]# echo $Fir_File_Lines  
36 anaconda-ks.cfg
```

要注意的是，`$()`仅在Bash Shell中有效，而反引号可在多种UNIX Shell中使用。所以这两种命令替换的方式各有特点，究竟选用哪种方式全看个人的喜好。

13.3 运算符

Shell中的运算符主要有比较运算符（用于整数比较）、字符串运算符（用于字符串测试）、文件操作运算符（用于文件测试）、逻辑运算符、算术运算符、位运算符、自增自减运算符等。本节将介绍后面3种运算符，即算术运算符、位运算符、自增自减运算符，其他几种运算符将在第14章中介绍。

13.3.1 算术运算符

算术运算符指的是加、减、乘、除、余、幂等常见的算术运算，以及加等、减等、乘等、除等、余等复合算术运算。要特别注意的是，Shell只支持整数计算，也就是所有可能产生小数的运算都会舍去小数部分。

常见的算术运算大多需要结合Shell的内建命令let来使用，在第11章中的“Shell的内建命令”部分已经有详细的例子，所以此处不赘述，仅列出表13-4和表13-5以供参考。请注意除法求余计算中，除数不能为0，否则Shell会报错。更多算术运算的方法请参考13.4节。

表13-4 常规算术运算符

运算符	运算符举例	运算结果
+ (加运算符)	1+1	2
- (减运算符)	2-1	1
* (乘运算符)	2*3	6
/ (除运算符)	9/4	2
% (余运算符)	10%3	1
** (幂运算符)	2**3	8

表13-5 复合算术运算符

运算符	运算符举例	变量 x 的运算结果
+= (加等运算符)	x=8;x+=2	10
-= (减等运算符)	x=8;x-=2	6
= (乘等运算符)	x=8;x=2	16
/= (除等运算符)	x=8;x/=2	4
%= (余等运算符)	x=8;x%=2	0

13.3.2 位运算符

位运算是基于内存中二进制数据的运算，也就是基于位的运算。常见的位运算有左移运算、右移运算、按位与、按位或、按位非、按位异或等运算。

位运算的左移、右移元素其实就是整数在内存中的“左右移动”，其中左移运算符为<<，右移运算符为>>。比如十进制整数4在内存中最后一个字节的排列如下：

0	0	0	0	0	1	0	0
---	---	---	---	---	---	---	---

如果对其进行左移2位的操作（左移后右边的空缺用0补足，下方粗体的0就是补足的部分），则在内存中变为十进制数16；相同的原理，如果对4做右移2位的操作，则值为1。

0	0	0	1	0	0	0	0
---	---	---	---	---	---	----------	----------

```
#
左移右移运算
#4
左移2
[root@localhost ~]# let "value=4<<2"
[root@localhost ~]# echo $value
16
#4
右移2
[root@localhost ~]# let "value=4>>2"
[root@localhost ~]# echo $value
1
```

按位与运算（&），是将两个整数写成二进制的形式，然后同位置相比较，只有当对应的二进制值都为1时，结果才为1。以8&4来说，计算方式如下，计算结果为0。

0	0	0	0	0	1	0	0
0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0

```
#  
按位与运算  
[root@localhost ~]# let "value=8&4"  
[root@localhost ~]# echo $value  
0
```

按位或运算（|），是将两个整数写成二进制的形式，然后同位置相比较，只要对应的位置有1，结果就为1。以8|4来说，其计算方式如下，计算结果为12。

0	0	0	0	0	1	0	0
0	0	0	0	1	0	0	0
0	0	0	0	1	1	0	0

```
#  
按位或运算
```

```
[root@localhost ~]# let "value=8|4"
[root@localhost ~]# echo $value
12
```

按位异或运算（ \wedge ），是将两个整数写成二进制的形式，然后同位置相比较，只要对应的位置同为0或同为1，结果就为0，否则为1。以 10^3 来说，其计算方式如下，计算结果为9。

0	0	0	0	1	0	1	0
0	0	0	0	0	0	1	1
0	0	0	0	1	0	0	1

```
#
按位异或运算
[root@localhost ~]# let "value=10^3"
[root@localhost ~]# echo $value
9
```

按位非（ \sim ）的计算方式比较麻烦，这里有个快捷的计算公式：“ $\sim a$ ”的值为“ $-(a+1)$ ”。

```
#
按位非运算
[root@localhost ~]# let "value=~8"
[root@localhost ~]# echo $value
-9
```

13.3.3 自增自减

自增自减运算主要包括前置自增、前置自减、后置自增、后置自减等。前置自增或前置自减操作会首先修改变量的值，然后再将变量的值传递出去；后置自增或后置自减则会首先将变量的值传递出去，然后再修改变量的值。自增符为“++”，自减符为“--”，操作对象只能是变量，不能是常数或表达式。如下所示：

```
[root@localhost ~]# cat add_minus.sh
#!/bin/bash
Add_01=10
Add_02=10
#Add_01
前置自增
#
也就是先将Add_01
自增1
变为11
，然后赋值给Add_03
，即为11
let "Add_03=(++Add_01)"
#Add_02
后置自增
#
也就是先将当前值赋给Add_04
，即10
，然后Add_02
自增1
，即为11
let "Add_04=(Add_02++)"
#
打印各变量的值
#
按照上面的计算方式，Add_01
、Add_02
、Add_03
为11
，Add_04
为10
echo Add_01 is:$Add_01
```

```
echo Add_02 is:$Add_02
echo Add_03 is:$Add_03
echo Add_04 is:$Add_04
[root@localhost ~]# bash add_minus.sh
Add_01 is:11
Add_02 is:11
Add_03 is:11
Add_04 is:10
```

13.4 其他算术运算

13.4.1 使用\$[]做运算

\$[]和\$(())类似，可用于简单的算术运算，以下给出使用方式：

```
[root@localhost ~]# echo ${1+1}
2
[root@localhost ~]# echo ${2-1}
1
[root@localhost ~]# echo ${2*2}
4
#
除法运算，由于是整数操作，舍去小数
[root@localhost ~]# echo ${5/2}
2
#
求余运算
[root@localhost ~]# echo ${5%2}
1
#
幂运算
[root@localhost ~]# echo ${5**2}
25
```

13.4.2 使用expr做运算

expr命令也可用于整数运算。和其他算术运算方式不同，expr要求操作数和操作符之间使用空格隔开（否则只会打印出字符串），所以特殊的操作符要使用转义符转义（比如*）。

expr支持的算术运算符有加、减、乘、除、余等，如下所示：

```
#
操作符和操作数之间没有空格则只会打印出字符串
[root@localhost ~]# expr 1+1
1+1
#
使用空格隔开后可以正常计算
[root@localhost ~]# expr 1 + 1
2
#
特殊符号（元字符）需要用转义符转义，否则出错
[root@localhost ~]# expr 2 * 2
expr: syntax error
[root@localhost ~]# expr 2 \* 2
4
```

13.4.3 内建运算命令declare

第11章中讲到declare是Shell的内建命令，通过它们也能进行整数运算。虽然说Shell可以不利用declare命令创建变量，但是通过下面的例子可以看到，它和显式使用declare定义变量的差别是很大的。在下面的示例中，例1里的变量I未经正式定义便赋值“1+1”，对Shell来说，此时的“1+1”只是一个字符串，和“abc”无异，所以打印出来也只是字符串。而例2中，使用declare显式地定义了整数变量J（-i参数指定变量为“整数”），此时再赋值“1+1”，Shell会将后面的字符串解析成算术运算，所以打印出的值是算术表达式的计算值。

```
#
例1
: 不使用declare
定义变量
[root@localhost ~]# I=1+1
[root@localhost ~]# echo $I
1+1
```

```
#
例2
: 使用declare
定义变量
[root@localhost ~]# declare -i J
[root@localhost ~]# J=1+1
[root@localhost ~]# echo $J
2
```

```
#
注意，Shell
中的算术运算要求运算符和操作数之间不能有空格，而是紧密连接的；特殊符号在
这里
也不需要转义（比如这里的+
号）；如果算术表达式中含有其他变量也不需要$
引用
```

13.4.4 算术扩展

算术扩展是Shell提供的整数变量的运算机制，是Shell的内建命令之一。其基本语法如下：

```
$(  
  算术表达式))
```

其中的算术表达式由变量和运算符组成，常见的用法是显示输出和变量赋值。若表达式中的变量没有定义，则在计算时，其值会被假设为0（但是并不会真的因此赋0值给该变量）。

```
#  
显示输出：  
echo $(  
  算术表达式))  
#  
例子：计算2*i+1  
的值  
[root@localhost ~]# i=2  
[root@localhost ~]# echo $((2*i+1)) #  
注意这里变量i  
前并没有$  
符  
5  
[root@localhost ~]# echo $((2*(i+1))) #  
用括号改变运算优先级  
6  
#  
变量赋值  
var=$(  
  算术表达式))  
#  
例子：将2*i+1  
的值赋值给变量var  
[root@localhost ~]# var=$((2*i+1))  
[root@localhost ~]# echo $var
```

5

#

未定义的变量参与算术表达式求值

```
[root@localhost ~]# echo $((2*(j+1)))
```

2

13.4.5 使用bc做运算

前面介绍的几种算术运算都只能是基于整数的，但现实中有很多需求必须是基于浮点数进行的计算，比如说职工的工资、贷款的利率等，这就要求更高精度的计算。而Linux下的bc正是这样一款专门用于高精度计算的工具。与其说bc是一个命令或者工具，不如说它是一门语言，bc的man文件对它的描述是：“一款高精度计算语言（An arbitrary precision calculator language）”。

在Linux下使用bc最简单的方式是直接输入bc命令，回车后进入bc的交互式界面，闪烁的命令提示符表明现在可以输入表达式了。注意看以下演示中的计算：

```
[root@localhost ~]# bc
bc 1.06
Copyright                                     1991-
1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
a=9
b=5
a+b
14
a-b
4
a*b
45
a/b
1
#
设置显示的小数位数
scale=3
a/b
1.800
```

在上面的示例中，先是定义了a=9，b=5，然后进行了加减

乘除计算，前面3次计算结果都是对的，唯独最后的除法计算不对：bc不是支持浮点计算吗？为什么看起来又是整数计算了？

事实上，默认情况下bc并不显示小数部分，必须设置要显示的小数位数。这可以通过设置scale做到，本例中设置为3后再运行除法运算结果就正确了。

除此之外，bc还支持逻辑运算、比较运算。

```
#
比较运算，真为1
，假为0
2>1
1
2<1
0
1==1
1
#
逻辑运算，真为1
，假为0
1&&2
1
1&&0
0
1||0
1
1||2
1
!0
1
```

上面介绍了bc的基本用法，但是在Shell编程中，往往只需要调用bc的处理结果而不会进入bc的交互界面。好在bc已经考虑到了这种情况，所以它支持批量的处理和以管道方式处理表达式计算。比如，希望一次性处理多个计算，只需要创建一个文件，并按行写好需要计算的表达式就可以了，如下所示：

```
[root@localhost ~]# cat cal.bc
12*34
34/12
scale=3;34/12
a=1;b=2;a+b
#
批量计算
[root@localhost ~]# cat cal.bc | bc
408
2
2.833
3
```

但有的时候需要在Shell程序中直接调用bc计算表达式，并将计算结果赋值给变量以参与后面的计算或判断。这里给出了一个求和的例子，如果想让脚本变得更灵活，也可以使用read命令动态地给变量赋值。

```
[root@localhost ~]# cat bc.sh
#!/bin/bash
NUM01=10
NUM02=15
TOTAL=$(echo "$NUM01+$NUM02" | bc)
echo $TOTAL
[root@localhost ~]# bash bc.sh
25
```

关于bc的更多用法可以参考man文件。正如本节一开始所说，bc更是一门语言，它有和常规编程语言一样的支持顺序执行、判断、循环等运行机制，还支持自定义函数等，功能非常强大，有兴趣的读者可以深入了解。

13.5 特殊字符

Shell中除了普通字符外，还有很多具有特殊含义和功能的字符，在使用它们时要特别注意其含义和作用。本节中有大部分内容都不是新知识，而是以往知识点的总结。

13.5.1 通配符

通配符用于模式匹配，常见的通配符有*、?和用[]括起来的字符序列。其中*代表任意长度的字符串。例如：a*可以匹配以a开头的任意长度的字符串，但是不包括点号和斜线号。也就是说a*不能匹配abc.txt。问号(?)可用于匹配任一单个字符。方括号[]代表匹配其中的任意一个字符，比如[abc]代表匹配a或者b或者c，[]中可以用-表明起止，比如[a-c]等同于[abc]，但是要注意-字符在[]外只是一个普通字符，没有任何特殊作用；*和?在[]中则变成了普通字符，没有通配的功效。

13.5.2 引号

引号包括单引号和双引号，单引号又叫称“全引用”或“强引用”；双引号又称“部分引用”或“弱引用”，所有用双引号括起来的字符除了美元符（\$）、反斜线（\）、反引号（`）依然保留其特殊用途外，其余字符都作为普通字符处理；而所有用单引号括起的部分都作为普通字符处理，但是要注意单引号中间不能再出现单引号，否则会Shell无法判断到底哪里是单引号的起止位置。

13.5.3 注释符

Shell使用#作为注释符。为了增强代码的可阅读性以及有利于后期管理，要养成多写注释的习惯。所有以#开头的部分Shell解释器都会略过。但是要注意，如果出现#后连着!，也就是“#!”不会被理解成注释，因此，其后跟着的部分必须是某个解释器的路径，而且“#!”必须出现在整个脚本的第一行。

13.5.4 大括号

1. 变量扩展

大括号{}在Shell中的用法很多，最常见的用法就是引用变量原型，又叫变量扩展，如表13-6所示。例如变量VAR，可以使用\${VAR}引用，这是推荐的引用变量的方法。

表13-6 大括号的变量扩展

表达式	作 用
<code>\${VAR}</code>	取出变量 VAR 的值
<code>\${VAR:-DEFAULT}</code>	如果 VAR 没有定义，则以 \$DEFAULT 作为其值
<code>\${VAR:=DEFAULT}</code>	如果 VAR 没有定义，或者值为空，则以 \$DEFAULT 作为其值
<code>\${VAR+VALUE}</code>	如果定义了 VAR，则值为 \$VALUE，否则为空字符串
<code>\${VAR:+VALUE}</code>	如果定义了 VAR 并且不为空值，则值为 \$VALUE，否则为空字符串
<code>\${VAR?MSG}</code>	如果 VAR 没有被定义，则打印 \$MSG
<code>\${VAR:?MSG}</code>	如果 VAR 没有被定义或未赋值，则打印 \$MSG
<code>\${!PREFIX*}</code> <code>\${!PREFIX@}</code>	匹配所有以 PREFIX 开头的变量
<code>\${#STR}</code>	返回 \$STR 的长度
<code>\${STR:POSITION}</code>	从位置 \$POSITION 处提取子串
<code>\${STR:POSITION:LENGTH}</code>	从位置 \$POSITION 处提取长度为 \$LENGTH 的子串
<code>\${STR#SUBSTR}</code>	从变量 \$STR 的开头处开始寻找，删除最短匹配 \$SUBSTR 的子串
<code>\${STR##SUBSTR}</code>	从变量 \$STR 的开头处开始寻找，删除最长匹配 \$SUBSTR 的子串
<code>\${STR%SUBSTR}</code>	从变量 \$STR 的结尾处开始寻找，删除最短匹配 \$SUBSTR 的子串
<code>\${STR%%SUBSTR}</code>	从变量 \$STR 的结尾处开始寻找，删除最长匹配 \$SUBSTR 的子串
<code>\${STR/SUBSTR/REPLACE}</code>	使用 \$REPLACE 替换第一个匹配的 \$SUBSTR
<code>\${STR//SUBSTR/REPLACE}</code>	使用 \$REPLACE 替换所有匹配的 \$SUBSTR
<code>\${STR/#SUBSTR/REPLACE}</code>	如果 \$STR 以 \$SUBSTR 开始，则用 \$REPLACE 来代替匹配到的 \$SUBSTR
<code>\${STR/%SUBSTR/REPLACE}</code>	如果 \$STR 以 \$SUBSTR 结束，则用 \$REPLACE 来代替匹配到的 \$SUBSTR

2.通配符扩展

用于匹配多个排列组合的可能。比如坐标，横坐标是x1、x2、x3，纵坐标是y1、y2、y3，那么所有可能的坐标就是

`{x1, x2, x3}{y1, y2, y3}`。

```
[root@localhost ~]# echo {x1,x2,x3}{y1,y2,y3}
x1y1 x1y2 x1y3 x2y1 x2y2 x2y3 x3y1 x3y2 x3y3
```

还可以用于匹配不同的文件，文件名的特征是只有其中一部分不同。比如file_A、file_B，就可以用file_{A,B}来匹配。

```
[root@localhost ~]# touch file_{A,B}
[root@localhost ~]# ls file_{A,B}
file_A  file_B
```

3. 语句块

大括号还能用于构造语句块，语句之间使用回车隔开。使用语句块的场景一般是在自定义函数中，本书将在第16章讲解函数的定义和使用。

13.5.5 控制字符

控制字符即Ctrl+KEY组合键一起使用，用于修改终端或文本显示。但是控制字符在脚本中不能使用，也就是说控制字符是交互式使用的。表13-7列出了常见的控制字符。

表13-7 控制字符

组合键	作 用
Ctrl+B	退格但是不删掉前面的字符
Ctrl+C	终结当前前台作业
Ctrl+D	结束符，可用于退出当前 Shell 或结束当前输入
Ctrl+G	系统输出一声鸣叫
Ctrl+H	退格且删掉前面的字符
Ctrl+L	清屏，和 clear 效果一样
Ctrl+I	水平制表符
Ctrl+K	垂直制表符
Ctrl+J	另起一行
Ctrl+M	回车
Ctrl+Z	暂停前台作业
Ctrl+V	在 vim 中操作 Visual Block
Ctrl+U	删除光标到行首的所有字符

13.5.6 杂项

1.反引号

反引号用于命令替换，和`$()`的作用相同，表示返回当前命令的执行结果并赋值给变量。

2.位置参数

位置参数的含义如下。

`$0`：脚本名本身。

`$1`、`$2`.....`${10}`：脚本的第一个参数、第二个参数.....第十个参数。

`$#`：变量总数。

`$*`、`$@`：显示所有参数。

`$?`：前一个命令的退出的返回值。

`!`：最后一个后台进程的ID号。

3.感叹号

通常代表逻辑反，例如`!=`代表不等于。也可以用于执行`history`中某个命令，比如使用`history`查看到第100个命令是`ifconfig`，则可以用`!100`代表执行`ifconfig`。

第14章 测试和判断

14.1 测试

程序运行中经常需要根据实际情况来运行特定的命令或代码段。比如，判断某个文件或目录是否存在，如果文件或目录不存在，可能需要首先创建文件或目录。举例说，要判断文件/var/log/message文件是否存在，可以先ls该文件，然后用\$?来判断，如下所示：

```
#ls
一个存在的文件
[root@localhost ~]# ls /var/log/messages
/var/log/messages
#
如果ls
成功，则$?
返回值为0
，说明该文件存在
[root@localhost ~]# echo $?
0
#ls
一个不存在的文件，命令本身会报错
[root@localhost ~]# ls /var/log/messages01
ls: /var/log/messages01: No such file or directory
#
这里$?
的返回值是非0
的，在不考虑文件权限的情况下，返回非0
值说明文件是不存在的
[root@localhost ~]# echo $?
2
```

上述的办法确实是一种办法，但是这意味着在很多情况下都需要自己来实现这个“判断”的过程，判断为真则返回0，为假则返回非0值。这种判断行为被称作“测试”。

实际上Shell已经实现了很多测试功能，这些测试语句不但使用起来非常简单，还能在少写代码的情况下实现同样的功能，最重要的是能让代码看起来更为清晰。

14.1.1 测试结构

测试的第一种使用方式是直接使用test命令，该命令的格式如下：

```
test expression
```

其中expression是一个表达式，可以是算术比较、字符串比较、文本和文件属性比较等。

第二种测试方式是使用“[”启动一个测试，再写expression，再以“]”结束测试。需要注意的是，左边的括号“[”后有个空格，右括号“]”前面也有个空格，如果任意一边少了空格都会造成Shell报错。为增加代码的可读性，推荐使用第二种方式，而且这种方式更容易与if、case、while这些条件判断的关键字连用，该测试结构如下：

```
[expression]
```

14.1.2 文件测试

本章开头提到的“测试文件是否存在”就是“文件测试”的一种典型需求。Shell中提供了大量的文件测试符，其格式如下：

```
#
文件测试方法一
test file_operator FILE
#
文件测试方法二
[ file_operator FILE ]
```

其中file_operator是文件测试符（具体参考表14-1），FILE是文件、目录（可以是文件或目录的全路径）。

同样以判断文件/var/log/message为例，使用文件测试的方法测试该文件“是否存在”，只需要使用-e操作符即可。

```
#
测试一个存在的文件则$?
返回0
[root@localhost ~]# test -e /var/log/messages
[root@localhost ~]# echo $?
0
#
测试一个不存在的文件$?
返回值不为0
[root@localhost ~]# test -e /var/log/messages01
[root@localhost ~]# echo $?
1
#
用[]
测试
[root@localhost ~]# [ -e /var/log/messages ]
[root@localhost ~]# echo $?
0
[root@localhost ~]# [ -e /var/log/messages01 ]
[root@localhost ~]# echo $?
```

表14-1罗列了Shell中的所有文件比较符，它们的使用方法可参照之前的例子，只需参考表格中的“文件测试”部分改变相应的文件操作符即可。

表14-1 文件测试符

文件测试	说 明
-b FILE	当文件存在且是个块文件时返回真，否则为假
-c FILE	当文件存在且是个字符设备时返回真，否则为假
-d FILE	当文件存在且是个目录时返回真，否则为假
-e FILE	当文件或者目录存在时返回真，否则为假
-f FILE	当文件存在且为普通文件时返回真，否则为假
-x FILE	当文件存在且为可执行文件时返回真，否则为假

(续)

文件测试	说 明
-w FILE	当文件存在且为可写文件时返回真，否则为假
-r FILE	当文件存在且为可读文件时返回真，否则为假
-l FILE	当文件存在且为连接文件时返回真，否则为假
-p FILE	当文件存在且为管道文件时返回真，否则为假
-s FILE	当文件存在且大小不为 0 时返回真，否则为假
-S FILE	当文件存在且为 socket 文件时返回真，否则为假
-g FILE	当文件存在且设置了 SGID 时返回真，否则为假
-u FILE	当文件存在且设置了 SUID 时返回真，否则为假
-k FILE	当文件存在且设置了 sticky 属性时返回真，否则为假
-G FILE	当文件存在且属于有效的用户组时返回真，否则为假
-O FILE	当文件存在且属于有效的用户时返回真，否则为假
FILE1 -nt FILE2	当 FILE1 比 FILE2 新时返回真，否则为假
FILE1 -ot FILE2	当 FILE1 比 FILE2 旧时返回真，否则为假

这里针对表14-1中最后两行内容——文件新旧的比较做一些说明。FILE1-nt FILE2中的-nt是newer than的意思，而FILE1-ot FILE2中的ot是older than的意思。文件新旧比较的主要使用场景是判断文件是否被更新或增量备份时，用于判断一段时间以来更新过的文件。

以下是参考表14-1写出的测试某个文件的读、写、执行属性的代码。

```
[root@localhost ~]# cat rwx.sh
#!/bin/bash
read -p "What file do you want to test?" filename
if [ ! -e "$filename" ]; then
    echo "The file does not exist."
    exit 1
fi
if [ -r "$filename" ]; then
    echo "$filename is readable."
fi
if [ -w "$filename" ]; then
    echo "$filename is writeable"
fi
if [ -x "$filename" ]; then
    echo "$filename is executable"
fi
```

14.1.3 字符串测试

Shell中的字符串比较主要有等于、不等于、大于、小于、是否为空等测试，如表14-2所示。

表14-2 字符串测试符

字符串测试	说 明
-z "string"	字符串 string 为空时返回真，否则为假
-n "string"	字符串 string 非空时返回真，否则为假
"string1" = "string2"	字符串 string1 和 string2 相同时返回真，否则为假
"string1" != "string2"	字符串 string1 和 string2 不相同时返回真，否则为假
"string1" > "string2"	按照字典排序，字符串 string1 排在 string2 之前时返回真，否则为假
"string1" < "string2"	按照字典排序，字符串 string1 排在 string2 之后时返回真，否则为假

下面演示了上述6个字符串测试符的用法，其中对str1的测试使用test方式，对str2的测试使用[]方式。

```
#
定义空字符串str1
[root@localhost ~]# str1=""
#
测试str1
是否为空，为空则返回0
[root@localhost ~]# test -z "$str1"
[root@localhost ~]# echo $?
0
#
测试str1
是否非空，非空则返回0
， 为空返回非0
， 此处返回1
[root@localhost ~]# test -n "$str1"
```

```
[root@localhost ~]# echo $?  
1  
#  
定义非空字符串str2  
， 值为hello  
[root@localhost ~]# str2="hello"  
#  
测试str2  
是否为空， 为空返回0  
， 不为空返回非0  
， 此处返回1  
[root@localhost ~]# [ -z "$str2" ]  
[root@localhost ~]# echo $?  
1  
#  
测试str2  
是否非空， 非空返回0  
[root@localhost ~]# [ -n "$str2" ]  
[root@localhost ~]# echo $?  
0  
#  
比较str1  
和str2  
是否相同， 相同则返回0  
， 否则返回非0  
， 此处返回1  
[root@localhost ~]# [ "$str1" = "$str2" ]  
[root@localhost ~]# echo $?  
1  
#  
比较str1  
和str2  
是否不同， 不同则返回0  
[root@localhost ~]# [ "$str1" != "$str2" ]  
[root@localhost ~]# echo $?  
0  
#  
比较str1  
和str1  
的大小， 需要注意的是， >  
和<  
都需要进行转义  
[root@localhost ~]# [ "$str1" \> "$str2" ]  
[root@localhost ~]# echo $?  
1  
[root@localhost ~]# [ "$str1" \< "$str2" ]  
[root@localhost ~]# echo $?  
0
```

#

如果不想用转义符，则可以用[[]]

括起表达式

```
[root@localhost ~]# [[ "$str1" > "$str2" ]]
```

```
[root@localhost ~]# echo $?
```

1

```
[root@localhost ~]# [[ "$str1" < "$str2" ]]
```

```
[root@localhost ~]# echo $?
```

0

14.1.4 整数比较

整数测试是一种简单的算术运算，作用在于比较两个整数的大小关系，测试成立则返回0，否则返回非0值。整数测试的格式如下：

```
#
整数测试方法一
test "num1" num_operator "num2"
#
整数测试方法二
[ "num1" num_operator "num2" ]
```

其中num_operator是整数测试符常见的整数测试符如表14-3所示。

表14-3 整数测试符

整数比较	说 明
"num1" -eq "num2"	如果 num1 等于 num2 则返回真，否则为假。其中 eq 为 equal
"num1" -gt "num2"	如果 num1 大于 num2 则返回真，否则为假。其中 gt 为 great than
"num1" -lt "num2"	如果 num1 小于 num2 则返回真，否则为假。其中 lt 为 less than
"num1" -ge "num2"	如果 num1 大于等于 num2 则返回真，否则为假。其中 ge 为 great equal
"num1" -le "num2"	如果 num1 小于等于 num2 则返回真，否则为假。其中 le 为 less equal
"num1" -ne "num2"	如果 num1 不等于 num2 则返回真，否则为假。其中 ne 为 not equal

下面是整数测试的演示示例，这里对定义的两个变量做了测试。

```
[root@localhost ~]# num1=10
[root@localhost ~]# num2=10
[root@localhost ~]# num3=9
[root@localhost ~]# num4=11
[root@localhost ~]# [ "$num1" -eq "$num2" ]
[root@localhost ~]# echo $?
0
[root@localhost ~]# [ "$num1" -gt "$num3" ]
[root@localhost ~]# echo $?
0
[root@localhost ~]# [ "$num1" -lt "$num4" ]
[root@localhost ~]# echo $?
0
[root@localhost ~]# [ "$num1" -ge "$num2" ]
[root@localhost ~]# echo $?
0
[root@localhost ~]# [ "$num1" -le "$num2" ]
[root@localhost ~]# echo $?
0
[root@localhost ~]# [ "$num1" -ne "$num3" ]
[root@localhost ~]# echo $?
0
```

14.1.5 逻辑测试符和逻辑运算符

逻辑测试用于连接多个测试条件，并返回整个表达式的值。逻辑测试主要有逻辑非、逻辑与、逻辑或3种。逻辑测试符如表14-4所示。

表14-4 逻辑测试符

逻辑运算	说 明
! expression	如果 expression 为真，则测试结果为假
expression1 -a expression2	expression1 和 expression2 同时为真，则测试结果为真
expression1 -o expression2	expression1 和 expression2 只要有一个为真，则测试结果为真

举例如下：

```
#
例一：逻辑非的使用
#
测试值为真的表达式在使用逻辑非后，表达式变为假，反之亦然
[root@localhost ~]# [ ! -e /var/log/messages ]
[root@localhost ~]# echo $?
1
#
例二：逻辑与的使用
#
表达式都为真，整个表达式才返回真，否则返回假
[root@localhost ~]# [ -e /var/log/messages -a -e /var/log/messages01 ]
[root@localhost ~]# echo $?
1
#
例三：逻辑或的使用
#
测试表达式中只要有真，则整个表达式返回真
[root@localhost ~]# [ -e /var/log/messages -o -e /var/log/messages01 ]
[root@localhost ~]# echo $?
```

如果读者曾经学过其他的编程语言，一定知道“逻辑运算符”也有逻辑非、逻辑与、逻辑或3种判断符号。事实上，在Shell中也有逻辑运算符，除了写法和使用方式不一样外，其作用和含义都是相同的，如表14-5所示。

表14-5 逻辑运算符

逻辑运算	说 明
!	逻辑非，对真假取反
&&	逻辑与，连接两个表达式，只有两个表达式为真结果才为真
	逻辑或，连接两个表达式，只要有一个表达式为真结果就为真

下面是一个使用逻辑运算符改写逻辑测试符的例子。通过此例，读者可以很容易地了解这两种方式在使用上的差异。

```
[root@localhost ~]# ! [ -e /var/log/messages ]
[root@localhost ~]# echo $?
1
[root@localhost ~]# [ -e /var/log/messages ] && [ -e /var/log/messages01 ]
[root@localhost ~]# echo $?
1
[root@localhost ~]# [ -e /var/log/messages ] || [ -e /var/log/messages01 ]
[root@localhost ~]# echo $?
0
```

不管是逻辑运算符还是逻辑测试符，在做逻辑与、逻辑或运算时都有共同的特点。比如逻辑与，由于需要所有的表达式都为0整体才返回0，因此在计算expression1结果为0后才会进行expression2的计算，如果expression2返回0则会进行

expression3的计算，如果在计算过程中任何一部分的计算值非0，则不会再计算后面的表达式。如果一开始就计算出expression1为非0，则可以断言整个表达式一定是返回非0，就没有必要计算expression2和expression3了。

```
#  
逻辑与运算  
expression1 && expression2 && expression3  
#  
逻辑与测试  
[ expression1 -a expression2 -a expression3 ]
```

而对逻辑或来说，只要有一个表达式返回0，则可以断言整个表达式的返回值是0，所以如果计算expression1的值为0，就不用再进行expression2和expression3的计算了。

```
#  
逻辑或运算  
expression1 || expression2 || expression3  
#  
逻辑或测试  
[ expression1 -o expression2 -o expression3 ]
```

在实践过程中，常会将逻辑与、逻辑或的这些特点联合起来使用，使用的方式如下：

```
expression && DoWhenExpressionTrue || DoWhenExpressionFalse
```

在这段代码中，从左到右分别用&&、||连接，这时，Shell首先计算expression，并返回其值。如果返回真，则会继续执行&&后的代码DoWhenExpressionTrue，如果该语句执行成功，则expression&&DoWhenExpressionTrue整体返回0，使用||

连接的DoWhenExpressionFalse代码将不会被执行；如果expression返回假，则跳过DoWhenExpressionTrue，这时由于expression&&DoWhenExpressionTrue整体返回假，则用||连接的代码段DoWhenExpressionFalse一定会被执行。由此可以看到，在保证DoWhenExpressionTrue一定能返回真的情况下，上述代码段其实就是一个隐形的if-then-else语法（这将会在下一小节介绍）。

14.2 判断

有了测试，就要有获得测试结果的机制，并根据测试结果运行不同的代码段，这样程序就可以从简单的命令罗列变得更“智能”一些，从而实现程序的流程控制。在Shell中，流程控制分为两大类，一类是“循环”，一类是“判断选择”。属于“循环”的有for、while、until，这将会在下一章中介绍，本节介绍“判断选择”，关键字是if、case。

14.2.1 if判断结构

if是最简单的判断语句，可以针对测试结果做相应处理：如果测试为真则运行相关代码，其语法结构如下：

```
if expression; then
    command
fi
```

如果expression测试返回真，则执行command。如果要执行的不止一条命令，则不同命令间用换行符隔开，如下所示：

```
if expression; then
    command1
    command2
    ...
fi
```

下面演示一个程序，该程序会根据输入的学生成绩打印对应的等级：大于等于80分的为A；大于等于60分、小于80分的为B、小于60分的为C。

```
[root@localhost ~]# cat score01.sh
#!/bin/bash
echo -n "Please input a score:"
read SCORE
if [ "$SCORE" -lt 60 ]; then
    echo "C"
fi
if [ "$SCORE" -lt 80 -a "$SCORE" -ge 60 ]; then
    echo "B"
fi
if [ "$SCORE" -ge 80 ]; then
    echo "A"
fi
```

#

脚本运行结果，依次输入95

、75

、45

时，脚本分别打印了正确的成绩等级

```
[root@localhost ~]# bash score01.sh
```

```
Please input a score:95
```

A

```
[root@localhost ~]# bash score01.sh
```

```
Please input a score:75
```

B

```
[root@localhost ~]# bash score01.sh
```

```
Please input a score:45
```

C

14.2.2 if/else判断结构

上一小节中的if结构非常简单，它只会在if判断为真的情况下执行then后面的内容，所以该语句只能做“单向选择”。虽然可以通过顺序使用多条if语句，以满足多种条件的判断，但是看起来还是比较烦琐。而if/else语句则可以完成两个分支的选择：如果if后的判断成立，则执行then后面的内容；否则执行else后面的内容。其语法结构如下：

```
if expression; then
    command
else
    command
fi
```

使用这种结构判断某个文件是否存在的示例如下：

```
#
检查文件是否存在
[root@localhost ~]# cat check_file.sh
#!/bin/bash
FILE=/var/log/messages
#FILE=/var/log/messages01
if [ -e $FILE ]; then
    echo "$FILE exists"
else
    echo "$FILE not exist"
fi
#
当FILE=/var/log/messages
时运行结果如下
[root@localhost ~]# bash check_file.sh
/var/log/messages exists
#
当FILE=/var/log/messages01
时运行结果如下
[root@localhost ~]# bash check_file.sh
```

/var/log/messages01 not exist

14.2.3 if/elif/else判断结构

不论是if结构的单向选择，还是if/else结构的双向选择，实际上都不能满足需要，现实中的判断往往有多种可能，在这种情况下可以通过if/else的语法嵌套完成多向选择。其结构如下所示：

```
if expression1; then
    command1
else
    if expression2; then
        command2
    else
        command3
    fi
fi
```

使用这种嵌套的方式可以增加更多的选择分支，虽然从语法上来说毫无错误，但使用这种方式进入多层嵌套后，代码的可读性会变得越来越差。下面使用if/else多层嵌套的方式将14.2.1小节中的演示代码改写成下面的格式：

```
[root@localhost ~]# cat score02.sh
#!/bin/bash
echo -n "Please input a score:"
read SCORE
if [ "$SCORE" -lt 60 ]; then
    echo "C"
else
    if [ "$SCORE" -lt 80 -a "$SCORE" -ge 60 ]; then #
        echo "B"
    else
        if [ "$SCORE" -ge 80 ]; then #
            echo "A"
        fi
    fi
fi
```

```
        fi
    fi
```

注意看改写后的结构，这里实现了3层嵌套。如果再进入更多的嵌套，相信读者会看得更加头痛。鉴于这种原因，并不建议使用if/else进行多层嵌套，而是使用if/elif/else结构，其语法结构如下：

```
if expression1; then
    command1
elif expression2; then
    Command2
elif expression3; then
    Command3
...
fi
```

这种结构可根据多种情况进行处理，而且看起来结构非常清晰。

下面使用if/elif/else结构来改写14.2.1小节中的示例代码，如下所示：

```
[root@localhost ~]# cat score03.sh
#!/bin/bash
echo -n "Please input a score:"
read SCORE
if [ "$SCORE" -lt 60 ]; then
    echo "C"
elif [ "$SCORE" -lt 80 -a "$SCORE" -ge 60 ]; then
    echo "B"
else
    echo "A"
fi
```

14.2.4 case判断结构

和if/elif/else判断结构一样，case判断结构也可以用于多种可能情况下的分支选择。其语法结构如下：

```
case VAR in
var1) command1 ;;
var2) command2 ;;
var3) command3 ;;
...
*) command ;;
esac
```

其原理为从上到下依次比较VAR和var1、var2、var3的值是否相等，如果匹配相等则执行后面的命令语句，在无一匹配的情况下匹配最后的默认*，并执行后面的默认命令。要注意的是，case判断结构中的var1、var2、var3等这些值只能是常量或正则表达式。

下面的脚本可以检测到当前操作系统类型。以下代码case中的匹配值是“常量”。

```
[root@localhost ~]# cat os_type.sh
#!/bin/bash
OS='uname -s'
case
"$OS"
" in
FreeBSD) echo "This is FreeBSD" ;;
CYGWIN_NT-5.1) echo "This is Cygwin" ;;
SunOS) echo "This is Solaris" ;;
Darwin) echo "This is Mac OSX" ;;
AIX) echo "This is AIX" ;;
Minix) echo "This is Minix" ;;
Linux) echo "This is Linux" ;;
*) echo "Failed to identify this OS" ;;
```

esac

下面的脚本可以用于检测用户的输入中是否含有大写字母、小写字母或者数字，这里case匹配的值是正则表达式。

```
[root@localhost ~]# cat detect_input.sh
#!/bin/bash
read -p "Give me a word: " input
echo -en "You gave me some "
case $input in
    *[:lower:]*) echo -en "Lowercase " ;;
    *[:upper:]*) echo -en "Uppercase " ;;
    *[:digit:]*) echo -en "Numerical " ;;
    *) echo "unknown input." ;;
esac
```

第15章 循环

在现实生活中，如果要某人不断地重复做某一件事情，那么他很快就会感觉到厌倦，并慢慢失去兴趣，随后效率也会渐渐降低，并越来越容易出错。而计算机在这方面就显得非常有“天赋”：它天生就适合做重复的事情，并乐此不疲。

在Shell编程中，很多时候需要反复执行一条或一组命令，比如说连续打印10条“Hello World”——虽然说不借助于循环而是写10条`echo "Hello World"`也可以完成同样的工作，但如果是打印100条呢？这时候如果还是采用这种写法，可能会让你烦不胜烦，这时就不得不借助于循环了。Shell中的循环主要有for、while、until、select几种。

15.1 for循环

for循环是Shell中最常见的循环结构，根据书写习惯又分为列表for循环、不带列表的for循环以及类C的for循环。for循环是一种运行前测试语句，也就是在运行任何循环体之前先要判断循环条件是否成立，只有在条件成立的情况下才会运行循环体，否则将退出循环。每完成一次循环后，在进行下一次循环之前都会再次进行测试。

15.1.1 带列表的for循环

带列表的for循环用于执行一定次数的循环（循环次数等于列表元素个数），其语法结构如下：

```
for VARIABLE in (list)
do
    command
done
```

下面的例子可循环打印出John喜爱的水果，每一次循环，变量FRUIT都被赋予了一个特定的值：由于列表中一共有4个元素，所以整个循环就会执行4次，每次循环变量FRUIT就依次被赋予一个列表中的值，所以第一次循环时FRUIT为apple，第二次为orange，第三次为banana，第四次为pear，执行到这里，将会随着列表的结束而停止循环体。一旦结束循环，脚本又会继续执行done后面的内容，本例中就是打印“No more fruits”。

```
[root@localhost ~]# cat fruit01.sh
#!/bin/bash
for FRUIT in apple orange banana pear
do
    echo "$FRUIT is John's favorite"
Done
echo "No more fruits"
#
```

运行结果

```
[root@localhost ~]# bash fruit.sh
apple is John's favorite
orange is John's favorite
banana is John's favorite
pear is John's favorite
No more fruits
```

上面脚本的写法并不是最好的，因为一旦列表元素改变了，你就不得不去改相应的for循环语句块。好的习惯是将列表定义为一个变量，然后在for中使用该变量。按照这种方法可以将上面的脚本修改成下面的形式：

```
[root@localhost ~]# cat fruit02.sh
#!/bin/bash
#
将列表定义到一个变量中，以后有任何修改只需要修改该变量即可
fruits="apple orange banana pear"
for FRUIT in ${fruits}
do
    echo "$FRUIT is John's favorite"
done
echo "No more fruits"
```

如果列表是数字，常规的方法是使用列表列出所有可能的数值，for循环会遍历所有列出的值。下面的脚本会循环5次并打印内容。

```
[root@localhost ~]# cat for_list01.sh
#!/bin/bash
for VAR in 1 2 3 4 5
do
    echo "Loop $VAR times"
done
#
执行结果
[root@localhost ~]# bash for_list01.sh
Loop 1 times
Loop 2 times
Loop 3 times
Loop 4 times
Loop 5 times
```

如果只是少数的几个数字还是比较方便一一枚举出来的，但是如果是1到100，这么写就不实际了，Shell提供了用于计数

的方式，比如说上例中1到5可以用{1..5}表示。所以脚本可以改写成下面的格式，运行结果和之前一致。

```
[root@localhost ~]# cat for_list02.sh
#!/bin/bash
for VAR in {1..5}
do
    echo "Loop $VAR times"
done
```

还可以使用seq命令结合命令替换的方式生成列表，下面的例子可以针对1到100的求和进行计算，其中的命令替换部分还可以使用\$()代替。

```
[root@localhost ~]# cat for_list03.sh
#!/bin/bash
sum=0
for VAR in `seq 1 100`
#for VAR in $(seq 1 100)
do
    let "sum+=VAR"
done
echo "Total: $sum"
#
运行结果
[root@localhost ~]# bash for_list03.sh
Total: 5050
```

下面是利用seq命令的“步长”计算1到100内的奇数和。

```
[root@localhost ~]# cat for_list04.sh
#!/bin/bash
sum=0
for VAR in $(seq 1 2 100)
do
    let "sum+=VAR"
done
```

```
echo "Total: $sum"  
[root@localhost ~]# bash for_list04.sh  
Total: 2500
```

从上面的命令替换的例子可以看出，其实列表for循环中in后面的内容可以是任意命令的标准输出。下面的例子中，会利用ls的输出作为in的列表，并循环打印所有文件的属性。

```
[root@localhost ~]# cat for_list05.sh  
#!/bin/bash  
for VAR in $(ls)  
do  
    ls -l $VAR  
done
```

15.1.2 不带列表的for循环

不带列表的for循环的结构如下所示：

```
for VARIABLE
do
    command
done
```

读者一定会诧异：既然没有列表，那么如何向这个for循环传递变量值呢？实际上，使用不带列表的for循环时，需要在运行脚本时通过参数的方式给for循环传递变量值。

```
[root@localhost ~]# cat for_list06.sh
#!/bin/bash
for VARIABLE
do
    echo -n "$VARIABLE "
done
echo
#
运行时向脚本传入参数
[root@localhost ~]# bash for_list06.sh 1 2 3
1 2 3
```

该语法虽然可以工作，但是可读性较差，所以不建议使用。可利用特殊变量\$@改写上述结构，使其变成下面的形式，功能是完全一样的。

```
[root@localhost ~]# cat for_list07.sh
#!/bin/bash
for VARIABLE in $@
do
    echo -n $VARIABLE
done
```

```
#  
运行时传入参数  
[root@localhost ~]# bash for_list07.sh 1 2 3  
1 2 3
```

15.1.3 类C的for循环

Shell支持类C的for循环。了解C语言或类C语言的读者一定会对(i=1;i<=10;i++)这样的结构十分熟悉，在Shell中其语法结构如下：

```
for ((expression1; expression2; expression3))
do
    command
done
```

其中，expression1为初始化语句，一般用作变量定义和初始化；expression2为判断表达式，用于测试表达式返回值并以此控制循环，返回值为真则循环继续，返回值为假时则退出循环；expression3用于变量值修改，从而影响expression2的返回值，并以此影响循环行为。下面的脚本演示了使用for语句控制的10次循环。

```
[root@localhost ~]# cat c_for01.sh
#!/bin/bash
for ((i=1; i<=10; i++))
do
    echo -n "$i "
done
echo
#
运行结果
[root@localhost ~]# bash c_for01.sh
1 2 3 4 5 6 7 8 9 10
```

使用类C的for循环还有其他的好处：可以在初始化expression1的同时初始化多个变量，另外，还可以在expression3中同时修改多个变量的值，每个expression中的多条

语句之间使用逗号隔开。示例如下：

```
[root@localhost ~]# cat c_for02.sh
#!/bin/bash
for ((i=1, j=100; i<=10; i++, j--))
do
    echo "i=$i j=$j "
done
[root@localhost ~]# bash c_for02.sh
i=1 j=100
i=2 j=99
i=3 j=98
i=4 j=97
i=5 j=96
i=6 j=95
i=7 j=94
i=8 j=93
i=9 j=92
i=10 j=91
```

下面是使用类C的for循环的示例，在该示例中同时计算了1到100的和以及1到100的奇数和。

```
[root@localhost ~]# cat c_for03.sh
#!/bin/bash
#sum01
用于计算1
到100
的和
#sum02
用于计算1
到100
的奇数和
sum01=0
sum02=0
for ((i=1, j=1; i<=100; i++, j+=2))
do
    let "sum01+=i"
    #
    由于j
    值增长速度比i
```

快，所以必须在过程中测试j

值不大于100

```
        if [ $j -lt 100 ]; then
            let "sum02+=j"
        fi
done
echo "sum01=$sum01"
echo "sum02=$sum02"
#
```

运行结果

```
[root@localhost ~]# bash c_for03.sh
```

```
sum01=5050
```

```
sum02=2500
```

15.1.4 for的无限循环

无限循环又叫“死循环”，要注意的是：和代码设计功能无关的无限循环，或者说是开发者意料之外的无限循环都属于软件bug，这类bug容易造成系统资源耗尽，造成严重的系统故障，所以要非常小心，避免出现这种问题。开发者在用循环语句的时候要尤其要注意循环结束条件，有条件的要进行测试。

使用类C的for循环结构可以很简单地制造无限循环，只需要保证expression2永远为真就可以了。下面的代码定义了i永远等于0，所以i<1永远成立，该代码会一直打印“infinite loop”，直至按下Ctrl+C组合键。

```
[root@localhost ~]# cat c_for04.sh
#!/bin/bash
for ((i=0; i<1; i+=0))
do
    echo "infinite loop"
done
```

更简单的方式是直接将条件表达式expression2置为1，而原本用于初始化的expression1和用户改变变量值的expression3则均置为空，代码如下所示：

```
[root@localhost ~]# cat c_for05.sh
#!/bin/bash
for ((;1;))
do
    echo "infinite loop"
done
```

15.2 while循环

15.2.1 while循环的语法

和for循环一样，while循环也是一种运行前测试语句，相比for循环来说，其语法更为简单，语法结构如下：

```
while expression
do
    command
done
```

首先while将测试expression的返回值，如果返回值为真则执行循环体，返回值为假将不执行循环。循环完成后进入下一次循环之前将再次测试。

如果已知循环次数，可以用计数的方式控制循环，即设定一个计数器，在达到规定的循环次数后退出循环。

```
[root@localhost ~]# cat while01.sh
#!/bin/bash
#
定义计数器，循环次数为5
CONTER=5
while [[ $CONTER -gt 0 ]] #
测试CONTER
的值大于0
的情况下继续循环
do
    echo -n "$CONTER "
    let "CONTER-=1" #
每次循环CONTER
值减1
done
echo
[root@localhost ~]# bash while01.sh
```

下面的示例使用类while循环，同时计算1到100的和以及1到100的奇数和。

```
[root@localhost ~]# cat while02.sh
#!/bin/bash
#sum01
用于计算1
到100
的和
#sum02
用于计算1
到100
的奇数和
sum01=0
sum02=0
i=1
j=1
while [[ "$i" -le "100" ]]
do
    let "sum01+=i"
    let "j=i%2" #
    变量j
    用来确定变量i
    的奇偶性，如是奇数则余为1
    if [[ $j -ne 0 ]]; then
        let "sum02+=i"
    fi
    let "i+=1"
done
echo "sum01=$sum01"
echo "sum02=$sum02"
#
运行结果
[root@localhost ~]# bash while02.sh
sum01=5050
sum02=2500
```

下面的示例是利用while做猜数字游戏，只有当输入的数字和预设的数字一致时，才会停止循环。

```
[root@localhost ~]# cat while03.sh
#!/bin/bash
PRE_SET_NUM=8
echo "Input a number between 1 and 10"
while read GUESS
do
    if [[ $GUESS -eq $PRE_SET_NUM ]]; then
        echo "You get the right number"
        exit
    else
        echo "Wrong, try again"
    fi
done
```


运行结果，输入6
和7
都提示输入错误，并要求继续输入

```
#
输入8
时，程序提示输入正确并退出循环
[root@localhost ~]# bash while03.sh
Input a number between 1 and 10
6
Wrong, try again
7
Wrong, try again
8
You get the right number
```

15.2.2 使用while按行读取文件

按行读取文件是while一个非常经典的用法，常用于处理格式化数据。比如说下面的一个用于记录学生信息的文件（读者自行创建，内容如下）。

```
[root@localhost ~]# cat student_info.txt
John    30      Boy
Sue     28      Girl
Wang    25      Boy
Xu      23      Girl
```

仔细观察这个文件的内容不难发现，第一列是姓名，第二列是年龄，第三列是性别。利用while可按行读取的特性，依次打印学生信息。

```
[root@localhost ~]# cat while04.sh
#!/bin/bash
while read LINE
do
    NAME=`echo $LINE | awk '{print $1}'`
    AGE=`echo $LINE | awk '{print $2}'`
    Sex=`echo $LINE | awk '{print $3}'`
    echo "My name is $NAME, I'm $AGE years old, I'm a $Se
done < student_info.txt
#
```

运行结果

```
[root@localhost ~]# bash while04.sh
My name is John, I'm 30 years old, I'm a Boy
My name is Sue, I'm 28 years old, I'm a Girl
My name is Wang, I'm 25 years old, I'm a Boy
My name is Xu, I'm 23 years old, I'm a Girl
```

上面采用输入重定向的方式完成了文件读取，使用管道也可以完成同样的效果，如下所示：

```
#while
使用管道的按行读取
[root@localhost ~]# cat while04.sh
#!/bin/bash
cat student_info.txt | while read LINE
do
    NAME=`echo $LINE | awk '{print $1}'`
    AGE=`echo $LINE | awk '{print $2}'`
    Sex=`echo $LINE | awk '{print $3}'`
    echo "My name is $NAME, I'm $AGE years old, I'm a $Sex"
done
```

虽然上面两段代码的功能看似一致，但这两种方式是有细微不同的：使用重定向符的while只会产生一个Shell，而使用管道的脚本在运行时会产生3个Shell，第一个Shell是cat（由于运行速度很快所以无法使用ps命令抓到），第二个Shell是管道，第三个Shell是while。

15.2.3 while的无限循环

和for循环相比，while的无限循环更为简单明了，主要有如下3种形式的写法。注意由while的无限循环结构本身并无终止循环的结构，所以要想跳出循环必须在循环体中自行判断，并使用循环控制语句break来终止循环（循环控制语句将在15.6小节中讲到）。

```
#
方法一
while ((1))
do
    command
done
#
方法二
while true
do
    command
done
#
方法三
while :
do
    command
done
```

我们可以利用while的无限循环实时的监测系统进程，以保证系统中的关键应用一直处于运行状态。

```
#!/bin/bash
while true
do
    HTTPD_STATUS=`service httpd status| grep running`
    if [ -z "$HTTPD_STATUS" ]; then
        echo "HTTPD is stopped, try to restart"
        service httpd restart
    else

```

```
                echo "HTTPD is running, wait 5 sec until next  
            fi  
            sleep 5  
done
```

15.3 until循环

15.3.1 until循环的语法

until循环也是运行前测试，但是until采用的是测试假值的方式，当测试结果为假时才继续执行循环体，直到测试为真时才停止循环。其语法如下：

```
until expression
do
    command
done
```

下面的示例使用until同时计算1到100的和以及1到100的奇数和。

```
[root@localhost ~]# cat until01.sh
#!/bin/bash
sum01=0
sum02=0
i=1
until [[ $i -gt 100 ]]
do
    let "sum01+=i"
    let "j=i%2"
    if [[ $j -ne 0 ]]; then
        let "sum02+=i"
    fi
    let "i+=1"
done
echo $sum01
echo $sum02
#
运行结果
[root@localhost ~]# bash until01.sh
5050
2500
```

15.3.2 until的无限循环

和while的无限循环相反，until的无限循环的条件是判断假成立时退出，其写法有如下两种：

```
#
方法一
until ((0))
do
    command
done
#
方法二
until false
do
    command
done
```

和while无限循环对比可发现，只要将while((1))改为until((0))，或将while true改为until false，就可以实现while和until无限循环的相互转换。

15.4 select循环

`select`是一种菜单扩展循环方式，其语法和带列表的`for`循环非常类似，基本结构如下：

```
select MENU in (list)
do
    command
done
```

当程序运行到`select`语句时，会自动将列表中的所有元素生成为可用1、2、3等数选择的列表，并等待用户输入。用户输入并回车后，`select`可判断输入并执行后续命令。如果用户在等待输入的光标后直接按回车键，`select`将不会退出而是再次生成列表等待输入。示例如下：

```
[root@localhost ~]# cat select01.sh
#!/bin/bash
echo "Which car do you prefer?"
select CAR in Benz Audi Volkswagen
do
    break #
这里用到了没有讲过的break
语句，这将在15.6
小节中讲到
done
echo "You chose $CAR"
#
运行结果
[root@localhost ~]# bash select01.sh
Which car do you prefer?
1) Benz
2) Audi
3) Volkswagen
#? #
此处尝试直接回车，结果select
再次生成了列表等待输入
```

```
1) Benz
2) Audi
3) VolksWagen
#?      2#
此处选择2
, 程序会退出select
并继续执行后面的语句
You chose Audi
```

通过上面的例子可以发现，select有判断用户输入的功能，所以select经常和case语句合并使用。

下面的例子使用select确认用户的输入并交由case处理，之后将根据不同输入执行不同代码段。代码中使用了“|”符，表示选择Saturday和Sunday的效果是一致的。

```
[root@localhost ~]# cat select02.sh
#!/bin/bash
select DAY in Mon Tue Wed Thu Fri Sat Sun
do
    case $DAY in
        Mon) echo "Today is Monday";;
        Tue) echo "Today is Tuesday";;
        Wed) echo "Today is Wednesday";;
        Thu) echo "Today is Thursday";;
        Fri) echo "Today is Friday";;
        Sat|Sun) echo "You can have a rest today";;
        *) echo "Unknown input, exit now" && break;;
    esac
done
#
运行结果
[root@localhost ~]# bash select02.sh
1) Monday      3) Wednesday  5) Friday      7) Sunday
2) Tuesday     4) Thursday   6) Saturday
#? 1
Today is Monday
#? 6
You can have a rest today
#? 7
You can have a rest today
#? 8
```

Unknown input, exit now

15.5 嵌套循环

所谓嵌套循环指的是一个循环语句中的循环体是另外一个循环。前面讲到的for、while、until、select循环语句都可以使用嵌套循环。在嵌套循环中可以多层嵌套，但是要注意，过度的嵌套会让程序变得晦涩难懂，所以除了确实必要的情况下，不建议使用多层嵌套（三层以上的嵌套）。

下面演示使用for的嵌套循环打印九九乘法表的方法。

```
[root@localhost ~]# cat nesting01.sh
#!/bin/bash
for ((i=1; i<=9; i++))
do
    for ((j=1; j<=9; j++))
    do
        let "multi=$i*$j"
        echo -n "$i*$j=$multi "
    done
    echo
done
```


运行结果

```
[root@localhost ~]# bash nesting01.sh
1*1=1 1*2=2 1*3=3 1*4=4 1*5=5 1*6=6 1*7=7 1*8=8 1*9=9
2*1=2 2*2=4 2*3=6 2*4=8 2*5=10 2*6=12 2*7=14 2*8=16 2*9=18
3*1=3 3*2=6 3*3=9 3*4=12 3*5=15 3*6=18 3*7=21 3*8=24 3*9=27
4*1=4 4*2=8 4*3=12 4*4=16 4*5=20 4*6=24 4*7=28 4*8=32 4*9=36
5*1=5 5*2=10 5*3=15 5*4=20 5*5=25 5*6=30 5*7=35 5*8=40 5*9=45
6*1=6 6*2=12 6*3=18 6*4=24 6*5=30 6*6=36 6*7=42 6*8=48 6*9=54
7*1=7 7*2=14 7*3=21 7*4=28 7*5=35 7*6=42 7*7=49 7*8=56 7*9=63
8*1=8 8*2=16 8*3=24 8*4=32 8*5=40 8*6=48 8*7=56 8*8=64 8*9=72
9*1=9 9*2=18 9*3=27 9*4=36 9*5=45 9*6=54 9*7=63 9*8=72 9*9=81
```

下面使用while改写九九乘法表，其运行结果和上面是一样的。

```
[root@localhost ~]# cat nesting02.sh
#!/bin/bash
i=1
while [[ "$i" -le "9" ]]
do
    j=1
    while [[ "$j" -le "9" ]]
    do
        let "multi=$i*$j"
        echo -n "$i*$j=$multi "
        let "j+=1"
    done
    echo
    let "i+=1"
done
```


运行结果

```
[root@localhost ~]# bash nesting02.sh
1*1=1 1*2=2 1*3=3 1*4=4 1*5=5 1*6=6 1*7=7 1*8=8 1*9=9
2*1=2 2*2=4 2*3=6 2*4=8 2*5=10 2*6=12 2*7=14 2*8=16 2*9=18
3*1=3 3*2=6 3*3=9 3*4=12 3*5=15 3*6=18 3*7=21 3*8=24 3*9=27
4*1=4 4*2=8 4*3=12 4*4=16 4*5=20 4*6=24 4*7=28 4*8=32 4*9=36
5*1=5 5*2=10 5*3=15 5*4=20 5*5=25 5*6=30 5*7=35 5*8=40 5*9=45
6*1=6 6*2=12 6*3=18 6*4=24 6*5=30 6*6=36 6*7=42 6*8=48 6*9=54
7*1=7 7*2=14 7*3=21 7*4=28 7*5=35 7*6=42 7*7=49 7*8=56 7*9=63
8*1=8 8*2=16 8*3=24 8*4=32 8*5=40 8*6=48 8*7=56 8*8=64 8*9=72
9*1=9 9*2=18 9*3=27 9*4=36 9*5=45 9*6=54 9*7=63 9*8=72 9*9=81
```

上面演示了for循环中嵌套for以及while循环嵌套while的写法，实际上不同循环之间也可以相互嵌套，比如for循环中嵌套while等。细心的读者可能已经发现，15.4节中的最后一个例子（select02.sh）其实就是一个select嵌套case的循环。

15.6 循环控制

15.6.1 break语句

break用于终止当前整个循环体。一般情况下，**break**都是和**if**判断语句一起使用的，当**if**条件满足时使用**break**终止循环。

15.5节中的九九乘法表看起来有点怪：有一半的内容是重复的，这时候就可以用**break**让程序更智能一点了。仔细观察可发现，内循环中的**j**的值会不断加1，只要**j**的值不超过**i**，内循环就会继续运行，即在内循环中判断**j**是否小于或等于**i**，返回真时继续循环，否则终止循环，这样打印出来的九九乘法表应该是一个三角形。按照这个思路将**for**循环的九九乘法表修改如下形式：

```
[root@localhost ~]# cat break01.sh
#!/bin/bash
for ((i=1; i<=9; i++))
do
    for ((j=1; j<=9; j++))
    do
        if [[ $j -le $i ]]; then #j
            小于等于i
            时运算
            let "multi=$i*$j"
            echo -n "$i*$j=$multi "
        else
            一旦大于i
            则立即停止当前循环
            break #j
        fi
    done
    echo
done
#
运行结果
```

```
[root@localhost ~]# bash break01.sh
1*1=1
2*1=2 2*2=4
3*1=3 3*2=6 3*3=9
4*1=4 4*2=8 4*3=12 4*4=16
5*1=5 5*2=10 5*3=15 5*4=20 5*5=25
6*1=6 6*2=12 6*3=18 6*4=24 6*5=30 6*6=36
7*1=7 7*2=14 7*3=21 7*4=28 7*5=35 7*6=42 7*7=49
8*1=8 8*2=16 8*3=24 8*4=32 8*5=40 8*6=48 8*7=56 8*8=64
9*1=9 9*2=18 9*3=27 9*4=36 9*5=45 9*6=54 9*7=63 9*8=72 9*9=81
```

读者可以尝试自行修改while的九九乘法表，并使运行结果和此处一致。

15.6.2 continue语句

`continue`语句用于结束当前循环转而进入下一次循环——注意，这是和`break`不同的地方：`continue`并不会终止当前的整个循环体，它只是提前结束本次循环，而循环体还将继续执行；而`break`则会结束整个循环体。下面的示例用`continue`打印了1到100之间的所有素数。根据素数的定义，素数只能被1和其自身整除。所以该程序应该用嵌套循环：外部循环是从1到100依次列举100个整数；内部循环是判断该数是否能被从2开始（包括2）到其本身的值为止（不包括本身）的数整除，如果存在这样的数，那么它就不是素数；不是素数就会立即`continue`到下一个数继续计算。

```
[root@localhost ~]# cat continue_02.sh
#!/bin/bash
for ((i=1; i<=100; i++))
do
    for ((j=2; j<i; j++))
    do
        if !(($i%$j)); then
            continue 2
        #continue
        后面的数字代表跳出循环的嵌套数，这里代表跳出了两层循环
    fi
done
echo -n "$i "
done
echo
#
运行结果
[root@localhost ~]# bash continue_02.sh
1 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73
```

第16章 函数

16.1 函数的基本知识

16.1.1 函数的定义和调用

函数是Shell脚本中自定义的一系列执行命令，一般来说函数应该设置有返回值（正确返回0，错误返回非0。对于错误返回，可以定义返回其他非0正值来细化错误，这将在下一节中详细描述）。使用函数最大的好处是可避免出现大量重复代码，同时增强了脚本的可读性：如果你在某个Shell脚本中看到checkFileExist这样的代码（实际上是函数调用），一定不难猜出代码的作用。

在Shell中定义函数的方法如下（其中function为定义函数的关键字，可以省略）：

```
#shell
中的函数定义
#
其中function
为关键字，FUNCTION_NAME
为函数名
function FUNCTION_NAME(){
    command1 #
    函数体中可以有多条语句，不允许有空语句
    command2
    ...
}
#
省略关键字function
，效果一致
FUNCTION_NAME(){
    command1
    command2
    ...
}
```

```
}
```

下面演示一个简单的函数定义和函数调用相关的例子。你可能会注意到，调用函数的方法时只要调用函数名即可。

```
[root@localhost ~]# cat sayHello.sh
#!/bin/bash
function sayHello(){      #
    定义函数sayHello
        echo "Hello"      #
    该函数的函数体为打印Hello
}                          #
    函数定义结束
echo "Call function sayHello"  #
    提示函数调用
sayHello                    #
    函数调用
#
    脚本执行结果
[root@localhost ~]# bash sayHello.sh
Call function sayHello
Hello                      #
    这里是调用函数的输出内容
```

下面的例子稍微复杂一点，在脚本中定义函数countLine，可计算指定文件的行数。

```
[root@localhost ~]# cat countLine.sh
#!/bin/bash
FILE=/etc/passwd          #
    指定要检查的文件
function countLine(){      #
    定义函数countLine
        local i=0
        while read line
        do
            let ++i
        done < $FILE
        echo "$FILE have $i lines"
    }
}
```

```
echo "Call function countLine"          #
提示函数调用
countLine                                #
函数调用
#
执行结果
[root@localhost ~]# bash countLine.sh
Call function countLine
/etc/passwd have 36 lines                #
这里是调用函数的输出内容
```

16.1.2 函数的返回值

函数的返回值又叫函数的退出状态，实际上是一种通信方式。举个生活中的例子便于大家更清楚地了解函数返回值的概念。

假设小王和同学小徐说好每周六早上10点都会到她家里玩，可是小王经常会迟到，这时候小徐都会发消息给小王问她出发了没有？小王收到消息后会根据实际情况回复消息，如果没出发就发送“NO”，否则发送“YES”。在这个例子中，小徐发消息问小王出发了没有，可以看作是一种“调用”，而小王的回复可以看作是调用的“返回值”。如果使用0代表“NO”、1代表“YES”，那么就更像真实的函数调用了。但是只有0和1这两种回复还是略显简单了些——如果是出发了，那么出发到哪里了？我们可以使用2代表走到1/4的路程、使用3代表走到1/2的路程、4代表走到3/4的路程、5代表已经到楼下等，这样返回的值就更有意义了。

Shell中的函数可以使用“返回值”的方式来给调用者反馈信息（使用return关键字），不要忘了获取上一个命令返回值的方式是使用\$?（关于\$?的用法参见13.1.6节）——这是获取函数返回值的主要方式。下面的例子中将创建checkFileExist函数，用于判断文件是否存在。

```
[root@localhost ~]# cat checkFileExist.sh
#!/bin/bash
FILE=/etc/notExistFile                                #
定义一个不存在的文件
function checkFileExist(){                             #
定义checkFileExist
函数
    if [ -f $FILE ]; then
        return 0
    else
```

```

        return 1
    fi
}
echo "Call function checkFileExist"      #
提示函数调用
checkFileExist                          #
调用函数
if [ $? -eq 0 ]; then
    echo "$FILE exist"
else
    echo "$FILE not exist"
fi
#
执行结果
[root@localhost ~]# bash checkFileExist.sh
Call function checkFileExist
/etc/notExistFile not exist             #
这里是调用函数的输出内容

```

下面举一个利用多个函数返回值判断用户输入的例子，如果用户输入的数值大于等于0且小于10则返回0，大于等于10且小于20则返回1，大于等于20且小于30则返回2，输入其余数值则返回3。

```

[root@localhost ~]# cat checkNum.sh
#!/bin/bash
function checkNum(){                      #
定义函数checkNum
    echo -n "Please input a number:"
    read NUM
    if [ $NUM -ge 0 -a $NUM -lt 10 ]; then
        return 0                          #
如果输入值属于[0,10)
则返回0
    fi
    if [ $NUM -ge 10 -a $NUM -lt 20 ]; then
        return 1                          #
如果输入值属于[10,20)
则返回1
    fi
    if [ $NUM -ge 20 -a $NUM -lt 30 ]; then
        return 2                          #
如果输入值属于[20,30)

```

```

    则返回2
        fi
        return 3                                #
    其余输入全部返回3
}
echo "Call function checkNum"                  #
提示函数调用
checkNum                                        #
函数调用
RTV=$?                                          #
将函数返回值保存到变量RTV
中
#
根据RTV
判断输入数据的范围
if [ $RTV -eq 0 ]; then
    echo "The number is between [0,10)"
elif [ $RTV -eq 1 ]; then
    echo "The number is between [10,20)"
elif [ $RTV -eq 2 ]; then
    echo "The number is between [20,30)"
else
    echo "Unknown input"
fi
#
脚本运行结果
[root@localhost ~]# bash checkNum.sh
Call function checkNum
Please input a number:5
The number is between [0,10)
[root@localhost ~]# bash checkNum.sh
Call function checkNum
Please input a number:15
The number is between [10,20)
[root@localhost ~]# bash checkNum.sh
Call function checkNum
Please input a number:25
The number is between [20,30)
[root@localhost ~]# bash checkNum.sh
Call function checkNum
Please input a number:35
Unknown input

```

16.2 带参数的函数

16.2.1 位置参数

在16.1.2节中，checkFileExist.sh脚本中定义了checkFileExist函数，但是可以看到这个脚本实际上写死了FILE变量，这会造成想要判断不同的文件是否存在时，需要修改脚本中的FILE变量——也就是要对代码本身的内容进行修改，这也是典型的代码和数据没有分开而导致的问题。事实上，可以通过定义带参数的函数解决这个问题。在Shell中，向函数传递参数也是使用位置参数来实现的。

使用带参数的函数修改之前的checkFileExist.sh脚本为checkFileExist_v2.sh，注意后面执行脚本时的传参方式。

```
[root@localhost ~]# cat checkFileExist_v2.sh
#!/bin/bash
function checkFileExist(){
    if [ -f $1 ]; then
        return 0
    else
        return 1
    fi
}
echo "Call function checkFileExist"
checkFileExist $1
if [ $? -eq 0 ]; then
    echo "$1 exist"
else
    echo "$1 not exist"
fi
#
# 执行脚本时，通过直接向脚本传递文件全路径的方式传递参数
#
# 可以看到这种方式不会因为测试文件的不一样而修改脚本本身的内容，非常简单
[root@localhost ~]# bash checkFileExist_v2.sh /etc/notExistFi
Call function checkFileExist
/etc/notExistFile not exist
```

```
[root@localhost ~]# bash checkFileExist_v2.sh /etc/passwd
Call function checkFileExist
/etc/passwd exist
```

下面的例子可以接受两个参数：\$1和\$2，该脚本将计算出\$1的\$2次方的值。

```
[root@localhost ~]# cat power.sh
#!/bin/bash
function power(){
    RESULT=1
    LOOP=0
    while [[ "$LOOP" -lt $2 ]]
    do
        let "RESULT=RESULT*$1"
        let "LOOP=LOOP+1"
    done
    echo $RESULT
}
echo "Call function power with parameters"
power $1 $2
#
计算2
的2
次方
[root@localhost ~]# bash power.sh 2 2
Call function power with parameters
4
#
计算3
的3
次方
[root@localhost ~]# bash power.sh 3 3
Call function power with parameters
27
```

16.2.2 指定位置参数值

除了在脚本运行时给脚本传入位置参数外，还可以使用内置命令set命令给脚本指定位置参数的值（又叫重置）。一旦使用set设置了传入参数的值，脚本将忽略运行时传入的位置参数（实际上是被set命令重置了位置参数的值）。

```
[root@localhost ~]# cat set01.sh
#!/bin/bash
set 1 2 3 4 5 6 #
设置脚本的6
个位置参数，其值分别是1 2 3 4 5 6
COUNT=1
for i in $@
do
    echo "Here \$$COUNT is $i"
    let "COUNT++"
done
#
运行结果如下
#
给脚本传入参数a b c d e f
，但是由于脚本运行时“重置”了位置参数的值，
所以打印出来的位置参数为脚本中设置的值
[root@localhost ~]# bash set01.sh a b c d e f
Here $1 is 1
Here $2 is 2
Here $3 is 3
Here $4 is 4
Here $5 is 5
Here $6 is 6
```

16.2.3 移动位置参数

在Shell中使用shift命令移动位置参数，第11章中曾简单讲到了在不加任何参数的情况下，shift命令可让位置参数左移一位，示例如下：

```
[root@localhost ~]# cat shift_03.sh
#!/bin/bash
until [ $# -eq 0 ]
do
    #
    打印当前的第一个参数$1
    , 和参数的总个数$#
    echo "Now $1 is: $1, total parameter is:$#"
    shift    #
    移动位置参数
done
#
运行结果
[root@localhost ~]# bash shift_03.sh a b c d
Now $1 is: a, total parameter is:4
Now $1 is: b, total parameter is:3
Now $1 is: c, total parameter is:2
Now $1 is: d, total parameter is:1
```

可以在shift命令后跟上向左移动的位数，比如说shift 2就是将位置参数整体向左移动两位。将上面的脚本修改一下后，运行结果如下：

```
#
如果将shift_03.sh
脚本中的shift
改为shift 2
, 则位置参数将会每次移动两位，运行结果如下
[root@localhost ~]# bash shift_03.sh a b c d
Now $1 is: a, total parameter is:4
Now $1 is: c, total parameter is:2
```

下面的例子是利用shift来计算脚本中所有参数的和。

```
[root@localhost ~]# cat shift_04.sh
#!/bin/bash
TOTAL=0
until [ $# -eq 0 ]
do
    let "TOTAL=TOTAL+$1"
    shift
done
echo $TOTAL
#
执行结果
[root@localhost ~]# bash shift_04.sh 10 20 30
60
```

16.3 函数库

对某些很常用的功能，必须考虑将其独立出来，集中存放在一些独立的文件中，这些文件就称为“函数库”。这么做的好处是在后期开发的过程中可以直接利用这些库函数写出高质量的代码。库函数的本质也是“函数”，所以它的定义方式和普通函数没有任何区别，但为了和一般函数区分开来，在实践中建议库函数使用下划线开头。

16.3.1 自定义函数库

由于Shell是一门面向过程的脚本型语言，而且用户主要是Linux系统管理人员，所以并没有非常活跃的社区，这也造成了Shell缺乏第三方函数库，所以在很多时候需要系统管理人员根据实际工作的需要自行开发函数库。下面建立一个叫lib01.sh的函数库，该函数库目前只有一个函数，用于判断文件是否存在。

```
[root@localhost ~]# cat lib01.sh
_checkFileExists(){
    if [ -f $1 ]; then
        echo "File:$1 exists"
    else
        echo "File:$1 not exist"
    fi
}
```

其他脚本在希望直接调用_checkFileExists函数时，可以通过直接加载lib01.sh函数库的方式实现。加载方式有如下两种：

```
#
# 使用“点”命令
[root@localhost ~]# . /PATH/TO/LIB
#
# 使用source
# 命令
[root@localhost ~]# source /PATH/TO/LIB
```

假设现在有个脚本想要直接调用_checkFileExists函数，可以通过加载lib01.sh函数库来实现。从下面的演示可以看出，通过调用函数库的方式会使开发脚本变得更为简便。

```
[root@localhost ~]# cat callLib01.sh
#!/bin/bash
source ./lib01.sh                                     #
引用当前目录下的lib01.sh
函数库
_checkFileExists /etc/notExistFile                  #
调用函数库中的函数
_checkFileExists /etc/passwd
#
执行结果
[root@localhost ~]# bash callLib01.sh
File:/etc/notExistFile not exist
File:/etc/passwd exists
```

16.3.2 函数库/etc/init.d/functions简介

很多Linux发行版中都有/etc/init.d目录，这是系统中放置所有开机启动脚本的目录，这些开机脚本在脚本开始运行时都会加载/etc/init.d/functions或/etc/rc.d/init.d/functions函数库（实际上这两个函数库的内容是完全一样的），如下所示：

```
# Source function library.  
. /etc/init.d/functions  
或者  
# Source function library.  
. /etc/rc.d/init.d/functions
```

为了让大家对functions函数有个初步的理解，在介绍functions函数库之前，先创建下面的脚本，并尝试运行。

```
[root@localhost ~]# cat callFunctions01.sh  
#!/bin/bash  
source /etc/init.d/functions  
confirm ITEM  
if [[ $? -eq 0 ]]; then  
    echo "ITEM confirmed"  
else  
    echo "ITEM not confirmed"  
fi  
#  
运行结果  
[root@localhost ~]# bash callFunctions01.sh  
Start service ITEM (Y)es/(N)o/(C)ontinue? [Y] Y  
ITEM confirmed  
[root@localhost ~]# bash callFunctions01.sh  
Start service ITEM (Y)es/(N)o/(C)ontinue? [Y] N  
ITEM not confirmed
```

从运行结果可以发现，脚本运行时会询问是否确认“Start service ITEM”，实际上这是functions函数库中confirm函数的功

能，如果用户输入“Y”确认，该函数会返回0值，否则返回非0。如此简单的一个函数调用不但会让脚本运行起来更为优雅，同时也不需要用户自行实现这样的功能，从而可以把精力更多地放在脚本本身的功能上，简化了开发过程。实际上 functions 函数库中定义了27个函数，表16-1中列举了常见的17个。

表16-1 functions函数库中的常用函数

checkpid()	检查某个 PID 是否存在
daemon()	以 daemon 方式启动某个服务
killproc()	停止某个进程
pidfileofproc()	检查某个进程的 PID 文件
pidofproc()	检查某个进程的 PID
status()	判断某个服务的状态
echo_success()	打印 OK
echo_failure()	打印 FAILED
echo_passed()	打印 PASSED
echo_warning()	打印 WARNING
success()	打印 OK 并记录日志
failure()	打印 FAILED 并记录日志
passed()	打印 PASSED 并记录日志

(续)

warning()	打印 WARNING 并记录日志
action()	执行给定的命令，并根据执行结果打印信息
strstr()	检查 \$1 字符串中是否含有 \$2 字符串
confirm()	提示是否启动某个服务

虽然说functions函数库为Linux管理员提供了一些好用的函数，但同时也可以看到，仅仅有这些函数还是远远不够的，所以自行开发函数库还是日常工作中很重要的部分。

16.4 递归函数

在说明什么是“递归函数”之前，让我们先讲一段小故事：1987年在陕西省宝鸡市的法门寺残塔中发现了一个神秘的盒子，现场考古人员打开后发现里面又是一个盒子，如此一共打开了8个盒子中的盒子，最终发现了一枚佛祖舍利（后证实是个假舍利）。如果把“打开盒子”这种行为当作是一次“函数调用”，这里就一共调用了8次，如果把整个过程从运行程序的角度来描述就是：调用“打开盒子”函数打开第一个盒子，如果发现里面还是盒子，则继续调用“打开盒子”函数打开第二只盒子，以此类推，直到不再有盒子为止——这种“类推”的方法用程序中的术语解释就是“递归”，具有“递归”功能的函数则被称为“递归函数”。递归函数的典型特征为：在函数体中继续调用函数自身。

那么这里就出现了一个问题，如果这种递归毫无止境地执行下去，是不是就成了“无限循环”了？答案是肯定的，所以递归函数一定要有结束递归的条件，当满足该条件时，递归就会终止。典型的递归函数的结构如下所示：

```
function recursion() {  
    recursion  
    conditionThatEndTheRecursion      #  
    停止递归的条件  
}
```

数学中有个经典的需要使用递归算法计算的公式是：阶乘。这里只讨论正整数阶乘的情况，任何大于1的自然数 n 阶乘的计算公式为： $n!=1\times 2\times 3\times \dots\times n$ ，或写成 $n!=n\times (n-1)!$ ，终止条件为 $0!=1$ 。按照该思路创建脚本factorial01.sh，内容如下：

```

[root@localhost ~]# cat factorial01.sh
#!/bin/bash
function factorial01() {
    local NUMBER=$1
    if [ $NUMBER -le 0 ]; then          #
        RES=1
    else
        factorial01 $((NUMBER-1))
        TEMP=$RES
        NUMBER=$NUMBER
        RES=$((NUMBER*TEMP))
    fi
}
factorial01 $1
echo $RES
#
使用该脚本计算指定正整数的阶乘
#
为了观察脚本运行过程，使用 -x
参数跟踪脚本的运行细节
[root@localhost ~]# bash -x factorial01.sh 6
+ factorial01 6
当NUMBER
为6
时，由于6
不等于0
，进入嵌套
+ local NUMBER=6
+ '[' 6 -le 0 ']'
+ factorial01 5 #
第一次嵌套时，NUMBER
为5
，继续进入嵌套
+ local NUMBER=5
+ '[' 5 -le 0 ']'
+ factorial01 4 #
第二次嵌套时，NUMBER
为4
，继续进入嵌套
+ local NUMBER=4
+ '[' 4 -le 0 ']'
+ factorial01 3 #
第三次嵌套时，NUMBER
为3
，继续进入嵌套
+ local NUMBER=3

```



```

+ '[' 3 -le 0 ']'
+ factorial01 2 #
第四次嵌套时，NUMBER
为2
，继续进入嵌套
+ local NUMBER=2
+ '[' 2 -le 0 ']'
+ factorial01 1 #
第五次嵌套时，NUMBER
为1
，继续进入嵌套
+ local NUMBER=1
+ '[' 1 -le 0 ']'
+ factorial01 0 #
第六次嵌套时，NUMBER
为0
，当前RES=1
+ local NUMBER=0
+ '[' 0 -le 0 ']'
+ RES=1
+ TEMP=1
+ NUMBER=1
+ RES=1 #
第六次嵌套的计算结果为1
，返回给第五次嵌套
+ TEMP=1
+ NUMBER=2
+ RES=2 #
第五次嵌套的计算结果为2
，返回给第四次嵌套
+ TEMP=2
+ NUMBER=3
+ RES=6 #
第四次嵌套的计算结果为6
，返回给第三次嵌套
+ TEMP=6
+ NUMBER=4
+ RES=24 #
第三次嵌套的计算结果为24
，返回给第二次嵌套
+ TEMP=24
+ NUMBER=5
+ RES=120 #
第二次嵌套的计算结果为120
，返回给第一次嵌套
+ TEMP=120
+ NUMBER=6
+ RES=720 #

```

```
第一次嵌套的计算结果为720  
， 计算结束  
+ echo 720  
720
```

递归另一个典型的例子是“汉诺塔”游戏。该游戏源自印度一个古老的传说：在印度北部的贝拿勒斯神庙中，一块黄铜板上插着三根宝石针，其中一根从下到上穿了由大到小的64个金盘片，这就是所谓的汉诺塔。僧侣们不分白天黑夜地按照下面的法则移动这些金盘片，但必须满足以下两个条件：

- 1) 一次只能移动一个盘片；
- 2) 所有宝石针上的盘片只能是小的在大的上面。

僧侣们预言，当所有的金盘片都从最初所在的那根宝石针上移动到另一根上的时候，便是世界的尽头。

实际上这只是一个传说。但我们可以用科学的算法算出这个游戏的复杂度为 $f(64)=2^{64}-1$ ，如果按照正确的方法移动盘片，并且可以做到每秒移动一次，那也要耗费5845亿年——届时可能真的是世界的尽头了。

为了简化汉诺塔游戏，这里只用4个盘片，如图16-1所示。我们的任务是将A柱上的4个盘片搬移到C柱上。将动作分解为以下几步：

- 第一步，将A柱上的3个盘片搬移到B柱上（递归搬移）；
- 第二步，将A柱上的一个盘片搬移到C柱上；
- 第三步，B柱上的3个盘片搬移到C柱上（递归搬移）。

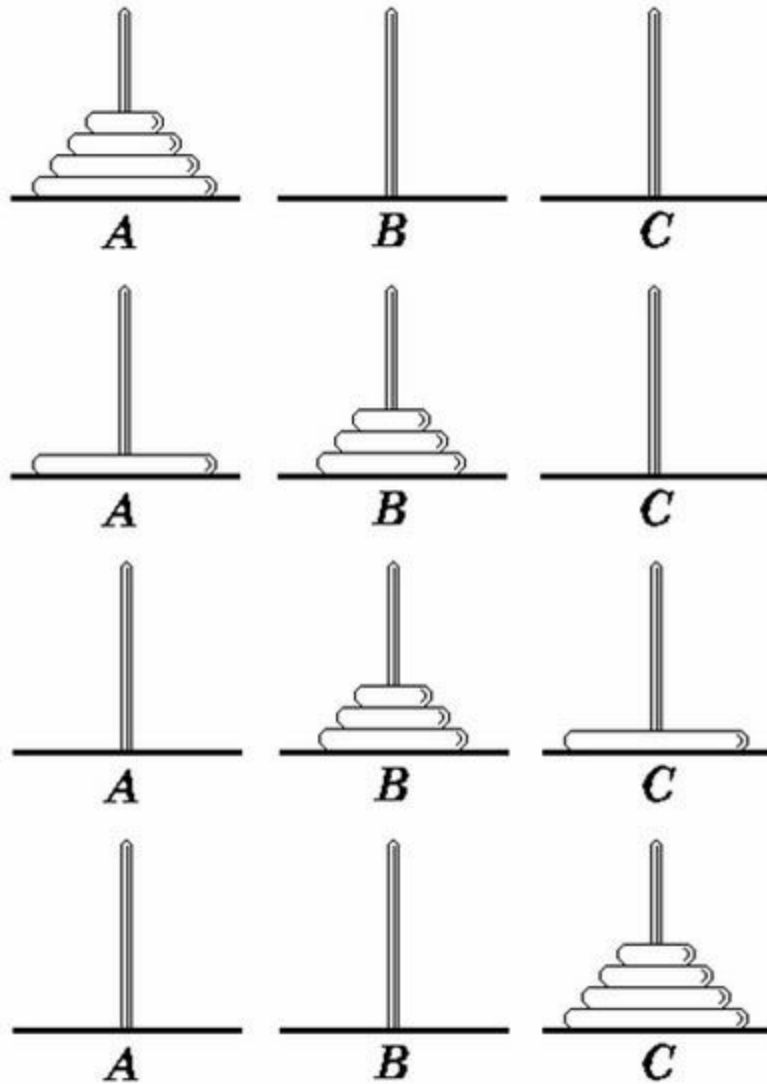


图16-1 汉诺塔

使用Shell脚本实现如下：

```
[root@localhost ~]# cat hanoi01.sh
#!/bin/bash
function hanoi01()
{
    local num=$1
    if [ "$num" -eq "1" ];then
        echo "Move:$2----->$4"
    else
        hanoi01 $((num-1)) $2 $4 $3
        echo "Move:$2----->$4"
    fi
}
```

```

        hanoi01 $((num-1)) $3 $2 $4
    fi
}
hanoi01 4 A B C #
将4
个盘片从A
柱上通过B
、C
柱移动到C
柱上
#
运行结果
[root@localhost ~]# bash hanoi01.sh
Move:A----->B
Move:A----->C
Move:B----->C
Move:A----->B
Move:C----->A
Move:C----->B
Move:A----->B
Move:A----->C
Move:B----->C
Move:B----->A
Move:C----->A
Move:B----->C
Move:A----->B
Move:A----->C
Move:B----->C

```

笔者想在结束本节之前谈谈个人对“递归函数”的看法：嵌套函数天生的结构就注定了其晦涩的可读性，在不少大公司内部的开发规范中也明确规定了不允许使用递归，所以在实际工作中要尽量避免使用递归。不过实际上你更可能根本没有使用递归的机会——从笔者多年从事Linux系统管理的经验来看，基本上不存在必须使用递归才能解决问题的场景。

第17章 重定向

17.1 重定向简介

17.1.1 重定向的基本概念

计算机最基础的功能是可以提供输入输出操作，常见的输入设备有键盘、鼠标、扫描仪等，对于Linux系统来说，通常以键盘为默认输入设备，又称标准输入设备；计算机常见的输出设备有显示器、蜂鸣器、打印机等，而Linux系统则以显示器为默认的输出设备，又称标准输出设备。所谓“重定向”，就是将原本应该从标准输入设备（键盘）输入的数据，改由其他文件或设备输入；或将原本应该输出到标准输出设备（显示器）的内容，改而输出到其他文件或设备上。

17.1.2 文件标识符和标准输入输出

文件标识符是重定向中很重要的一个概念，Linux使用0到9的整数指明了与特定进程相关的数据流，系统在启动一个进程的同时会为该进程打开三个文件：标准输入（`stdin`）、标准输出（`stdout`）、标准错误输出（`stderr`），分别用文件标识符0、1、2来标识。如果要为进程打开其他的输入输出，则需要从整数3开始标识。默认情况下，标准输入为键盘，标准输出和错误输出为显示器。

17.2 I/O重定向

17.2.1 I/O重定向符号和用法

I/O重定向是重定向中的一个重要部分，在Shell编程中会有很多机会用到这个功能。简单来说，I/O重定向可以将任何文件、命令、脚本、程序或脚本的输出重定向到另外一个文件、命令、程序或脚本。

I/O重定向常见符号和功能描述如表17-1所示。

表17-1 常见的I/O重定向符号

符号	含 义
>	标准输出覆盖重定向：将命令的输出重定向输出到其他文件中
>>	标准输出追加重定向：将命令的输出重定向输出到其他文件中，同时会覆盖文件中的已有内容
>&	标识输出重定向：将一个标识的输出重定向到另一个标识的输入
<	标准输入重定向：命令将从指定文件中读取输入而不是从键盘输入
	管道：从一个命令中读取输出并作为另一个命令的输入

读者初次看到表17-1一定会感觉不知所云，所以下面将逐一介绍上述符号的具体使用方法，建议读者跟随所讲的内容动手实践，以加深理解。

1.标准输出覆盖重定向：>

使用标准输出覆盖重定向符号可以将原本输出到显示器上的内容重定向到一个文件中，比如使用ls-l可以列出指定目录中文件的详细信息，但是如果想把结果保存到文件中以便日后查看，则可以使用标准输出覆盖重定向符。示例如下：

```
#
列出/usr
目录中的文件信息，默认所产生的输出会显示在屏幕上
[root@localhost ~]# ls -l /usr/
total 176
drwxr-xr-x  2 root root 40960 Apr 11 11:19 bin
drwxr-xr-x  2 root root  4096 Oct  1  2009 etc
drwxr-xr-x  2 root root  4096 Oct  1  2009 games
drwxr-xr-x 50 root root  4096 Apr 11 10:21 include
drwxr-xr-x  6 root root  4096 Feb 23  2012 kerberos
drwxr-xr-x 99 root root 36864 Apr 11 11:19 lib
drwxr-xr-x 11 root root  4096 Apr 11 11:19 libexec
drwxr-xr-x 12 root root  4096 Apr 11 10:38 local
drwxr-xr-x  2 root root 12288 Apr 11 11:19 sbin
drwxr-xr-x 183 root root  4096 Apr 11 10:21 share
drwxr-xr-x  4 root root  4096 Feb 26 21:13 src
lrwxrwxrwx   1 root root      10 Feb 26 21:13 tmp -
> ../var/tmp
drwxr-xr-x  3 root root  4096 Feb 26 21:15 X11R6
[root@localhost ~]# ls -l /usr/ > ls_usr.txt #
回车
[root@localhost ~]# #
注意到回车后并没有任何输出，因为输出被重定向到文件中
[root@localhost ~]# cat ls_usr.txt #
此文件内容和之前的输出一致
total 176
drwxr-xr-x  2 root root 40960 Apr 11 11:19 bin
drwxr-xr-x  2 root root  4096 Oct  1  2009 etc
drwxr-xr-x  2 root root  4096 Oct  1  2009 games
drwxr-xr-x 50 root root  4096 Apr 11 10:21 include
drwxr-xr-x  6 root root  4096 Feb 23  2012 kerberos
drwxr-xr-x 99 root root 36864 Apr 11 11:19 lib
drwxr-xr-x 11 root root  4096 Apr 11 11:19 libexec
drwxr-xr-x 12 root root  4096 Apr 11 10:38 local
drwxr-xr-x  2 root root 12288 Apr 11 11:19 sbin
drwxr-xr-x 183 root root  4096 Apr 11 10:21 share
drwxr-xr-x  4 root root  4096 Feb 26 21:13 src
lrwxrwxrwx   1 root root      10 Feb 26 21:13 tmp -
> ../var/tmp
drwxr-xr-x  3 root root  4096 Feb 26 21:15 X11R6
```

请注意，如果指定的重定向文件不存在，则命令会先创建这个文件，如果文件存在且内容不为空，则原文件内容将被全

部清空。所以有时候需要先判断该文件是否存在，以避免不小心破坏了原有文件内容。

下面尝试ls一个不存在的文件，看看会发生什么。

```
[root@localhost ~]# ls -l /usr/noExist > ls_noExist.txt
ls: /usr/noExist: No such file or directory
```

这里ls命令发现指定的文件不存在后给出了错误输出。这是为什么呢？

标准输出覆盖重定向其实是默认将文件标识符为1的内容重定向到指定文件中，所以如下两种写法是等价的，或者说，第一种写法是第二种写法的“省略写法”。

```
#
标准输出覆盖重定向符默认只将文件标识符为1
的内容重定向到指定文件
[root@localhost ~]# ls -l /usr/ > ls_usr.txt
#
以上写法等价于
[root@localhost ~]# ls -l /usr/ 1> ls_usr.txt
```

如果命令由于各种原因出错时所产生的错误输出，其文件标识符为2，而标准错误的输出默认也是显示器。所以我们可以指定将文件标识符为2的内容重定向到指定文件，这样错误输出就不会出现在显示器上了。如下所示：

```
[root@localhost ~]# ls -l /usr/noExist 2> ls_noExist_err.txt#
错误输出被重定向到文件中
[root@localhost ~]# cat ls_noExist_err.txt
ls: /usr/noExist: No such file or directory
```

如果某命令的输出既有标准输出，又有标准错误输出，则可以分别指定不同标识符的内容输出到不同的文件中。

```
[root@localhost ~]# COMMAND 1> stdout.txt 2>stderr.txt
```

2.标准输出追加重定向：>>

该符号用法和>完全一致，不同的只是如果指定的重定向文件存在且内容不为空，重定向并不会清空原文件内容，而是将命令的输出新增到原文件的尾部。在下面的例子中，先后将/usr和/tmp目录中列出的内容追加重定向到append.txt文件。

```
[root@localhost ~]# ls -l /usr/ >> append.txt
[root@localhost ~]# ls -l /tmp/ >> append.txt
[root@localhost ~]# cat append.txt
total 176
drwxr-xr-x  2 root root 40960 Apr 11 11:19 bin
drwxr-xr-x  2 root root  4096 Oct  1  2009 etc
drwxr-xr-x  2 root root  4096 Oct  1  2009 games
drwxr-xr-x 50 root root  4096 Apr 11 10:21 include
drwxr-xr-x  6 root root  4096 Feb 23  2012 kerberos
drwxr-xr-x 99 root root 36864 Apr 11 11:19 lib
drwxr-xr-x 11 root root  4096 Apr 11 11:19 libexec
drwxr-xr-x 12 root root  4096 Apr 11 10:38 local
drwxr-xr-x  2 root root 12288 Apr 11 11:19 sbin
drwxr-xr-x 183 root root  4096 Apr 11 10:21 share
drwxr-xr-x  4 root root  4096 Feb 26 21:13 src
lrwxrwxrwx  1 root root      10 Feb 26 21:13 tmp -
> ../var/tmp
drwxr-xr-x  3 root root  4096 Feb 26 21:15 X11R6
total 0
srwxr-xr-x 1 root root 0 Mar  1 11:27 gedit.root.903135227
srwxr-xr-x 1 root root 0 Mar 28 00:18 mapping-root
```

3.标识输出重定向：>&

标识输出重定向的作用是将一个标识的输出重定向到另一

个标识的输入。比如想要将标准输出和标准错误同时定向到同一个文件中，可使用如下命令：

```
[root@localhost ~]# COMMAND > stdout_stderr.txt 2>&1
```

上面演示的命令从左到右可以读为：执行COMMAND命令，将标准输出的内容重定向到stdout_stderr.txt中，如果有标准错误输出也同时重定向到该文件中。

为演示执行某个命令后既有标准输出又有错误输出的情景，可以使用普通用户执行以下命令，从输出内容可以看到，除了标准输出外还有很多由于文件权限问题而出现的错误输出。

```
[user1@localhost ~]$ find / -type f -name *.txt
find: /var/log/httpd: Permission denied
find: /var/log/ppp: Permission denied
find: /var/log/audit: Permission denied
find: /var/log/samba: Permission denied
find: /proc/11771/task/11771/fd: Permission denied
find: /proc/11771/fd: Permission denied
find: /proc/11951/task/11951/fd: Permission denied
find: /proc/11951/fd: Permission denied
find: /proc/11953/task/11953/fd: Permission denied
find: /proc/11953/fd: Permission denied
find: /proc/11985/task/11985/fd: Permission denied
find: /proc/11985/fd: Permission denied
find: /proc/12021/task/12021/fd: Permission denied
.....(
略去内容).....
/usr/lib/python2.4/email/test/data/msg_16.txt
/usr/lib/xulrunner-1.9/README.txt
find: /usr/lib/audit: Permission denied
/usr/lib/firefox-3.0.18/README.txt
```

现试图将上述所有输出重定向到某个文件中，使用以下命

令只能将标准输出覆盖重定向到find_res01.txt文件，而大量的错误输出依然会出现在显示器上。

```
[user1@localhost ~]$ find / -type f -name *.txt > find_res01.txt
find: /var/log/httpd: Permission denied
find: /var/log/ppp: Permission denied
find: /var/log/audit: Permission denied
find: /var/log/samba: Permission denied
find: /var/empty/sshd: Permission denied
find: /var/run/cups/certs: Permission denied
```

下面使用标识输出重定向，将标准错误输出同时定向到find_res01.txt文件。

```
[user1@localhost ~]$ find / -type f -name *.txt > find_res01.txt 2>&1
[user1@localhost ~]$ #
屏幕上看不到任何输出
```

很多时候大家并不会在意错误输出，特别是一些系统后台任务可能在每天凌晨运行，这时出现的错误输出可能并不是系统管理员所关心的，所以也没有必要将错误输出保存到任何文件中。这时可以利用系统中的一个特殊设备/dev/null，将所有错误输出重定向到该设备中——系统会将任何输入到该设备的内容全部删除（就像一个宇宙黑洞）。

```
[root@localhost ~]# COMMAND > stdout.txt 2> /dev/null
```

4.标准输入重定向：<

标准输入重定向可以将原本应由从标准输入设备中读取的

内容转由文件内容输入，也就是将文件内容写入标准输入中。

下面的例子中首先运行cat命令，系统将等待键盘输入。如果此时输入Hello并回车，cat命令会读取并立即输出Hello，然后命令将继续等待输入，直到使用Ctrl+D组合键终止输入。

```
[root@localhost ~]# cat
Hello    #
从键盘输入Hello
Hello    #cat
命令读取并输出Hello
World    #
从键盘输入World
World    #cat
命令读取并输出World
[Ctrl+D]
终止输入
[root@localhost ~]#
```

在下面的例一中，先将要打印的内容提前写到某个文件中，比如HelloWorld01.txt，然后使用标准输入重定向将内容重定向给cat命令。从命令输出结果可以看出，文件内容被标准输入重定向给了cat命令，然后该命令忠实地履行了打印任务。

```
#
例一：给cat
的标准输入重定向
[root@localhost ~]# cat HelloWorld01.txt
Hello
World
[root@localhost ~]# cat < HelloWorld01.txt
Hello
World
```

下面是使用sort排序的例子，运行sort命令后系统将等待键

盘输入，依次输入3个单词并以回车符隔开，输入结束后使用Ctrl+D组合键终止输入。此时sort命令会打印出排序后的单词列表。

```
[root@localhost ~]# sort
banana
apple
carrot
[Ctrl+D]
终止输入
apple
banana
carrot
[root@localhost ~]#
```

如果将需要排序的单词预先写到fruit01.txt文件中，然后使用标准输入重定向给sort命令，效果和之前一致，如例二所示：

```
#
例二：给sort
的标准输入重定向
[root@localhost ~]# cat fruit01.txt
banana
apple
carrot
[root@localhost ~]# sort < fruit01.txt
apple
banana
carrot
```

5.管道：|

管道也是一种重要的I/O重定向方法，在第5章中我们已经介绍过管道的概念和基本用法。简单地说管道就是将一个命令的输出作为另一个命令的输入，借此方式可通过多个简单命令

的共同协作来完成较为复杂的工作。读者可以行复习第5章来加强对管道的理解。

17.2.2 使用exec

exec是Shell的内建命令，执行这个命令时系统不会启动新的Shell，而是用要被执行的命令替换当前的Shell进程。因此假设在一个Shell中执行exec ls，则在列出当前目录后该Shell进程将会主动退出——如果使用ssh进行远程连接，则当前连接也会在执行完这个命令后断开。除此之外，exec还可以用于I/O重定向，表17-2总结了exec的用法。

表17-2 exec的常见用法

命 令	说 明
exec <file	将 file 文件中的内容作为 exec 的标准输入
exec >file	将 file 文件作为标准输出
exec 3<file	指定文件标识符
exec 3<&-	关闭文件标识符
exec 3>file	将写入指定文件标识符的内容写入指定文件（这里的文件是 file）
exec 4<&3	创建文件标识符 3 的拷贝 4

1.将file文件中的内容作为exec的标准输入

创建文件command.txt，该文件中罗列了需要执行的命令，然后将该文件重定向为exec的标准输入，可以看到系统成功执行了文件中的所有命令。但是在命令退出后同时断开了终端，正如之前所说，系统调用exec是以新的进程去代替原来的进程，但进程的PID保持不变。所以exec并不会创建新的进程，只是替换了原来进程上下文的内容，因此当被执行的命令退出时，这个进程也随之退出了。示例如下：

```
[root@localhost ~]# cat command.txt
echo "Hello"
echo "World"
[root@localhost ~]# exec <command.txt
[root@localhost ~]# echo "Hello"
Hello
[root@localhost ~]# echo "World"
World
[root@localhost ~]# logout
```

2.将file文件作为标准输出

利用exec可将标准输出全部重定向到某个文件中。下面例子中的第一条命令，即将本次会话中的所有标准输出重定向到了out01.txt文件中。你可能已注意到第二条、第三条命令均没有任何输出到屏幕，第四条命令exec>/dev/tty是将标准输出重新定向到了显示器（/dev/tty为显示终端）上。最后查看out01.txt文件内容，正是之前两条命令的输出内容。

```
[root@localhost ~]# exec >out01.txt
[root@localhost ~]# pwd
[root@localhost ~]# echo "HelloWorld"
[root@localhost ~]# exec >/dev/tty
[root@localhost ~]# cat out01.txt
/root
HelloWorld
```

3.指定文件标识符

通过exec指定文件标识符，可以通过对该标识符的操作进行文件的操作。还是拿排序为例，下面会读入fruit01.txt文件并指定标识符为3，然后对其进行排序。

```
[root@localhost ~]# cat fruit01.txt
banana
```

```
apple
carrot
[root@localhost ~]# exec 3<fruit01.txt
[root@localhost ~]# sort <&3
apple
banana
carrot
```

实际上，文件标识符类似于很多编程语言中的“句柄”。在Linux中，文件标识符也是一种“设备”，这个标识符会出现在/dev/fd目录中。进入这个目录，可以看到3是一个到该文件的软链接——记住：Linux下一切皆文件。

```
[root@localhost ~]# cd /dev/fd
[root@localhost fd]# ll
total 0
lrwx----- 1 root root 64 May 30 03:40 0 -> /dev/pts/1
lrwx----- 1 root root 64 May 30 03:54 1 -> /dev/pts/1
lrwx----- 1 root root 64 May 30 03:54 2 -> /dev/pts/1
lrwx----- 1 root root 64 May 30 03:54 255 -> /dev/pts/1
lr-x----- 1 root root 64 May 30 03:54 3 -
> /root/fruit01.txt
```

4.关闭文件标识符

主动打开的文件标识符需要主动关闭，否则除了系统重启，该文件标识符会一直被占用。关闭文件标识符的方法如下：

```
[root@localhost fd]# exec 3<&-
[root@localhost fd]# ls -l
total 0
lrwx----- 1 root root 64 May 30 05:15 0 -> /dev/pts/0
lrwx----- 1 root root 64 May 30 05:39 1 -> /dev/pts/0
lrwx----- 1 root root 64 May 30 05:39 2 -> /dev/pts/0
lrwx----- 1 root root 64 May 30 05:39 255 -> /dev/pts/0
```

完成后使用ls命令确认3已经消失。

5.将写入指定文件标识符的内容写入指定文件

命令及示例如下：

```
#
创建文件标识符3
，并将写入3
的内容全部写入file
文件中
[root@localhost ~]# exec 3>file
#
命令的输出内容写到标识符3
中
[root@localhost ~]# echo "HelloWorld" >&3
[root@localhost ~]# cat file
HelloWorld
```

6.创建文件标识符的拷贝

命令及示例如下：

```
[root@localhost ~]# exec 3<fruit01.txt
#
创建文件标识符3
的拷贝4
[root@localhost ~]# exec 4<&3
#
可以看到3
和4
都是指向同一个文件
[root@localhost ~]# ls -l /dev/fd/
total 0
lrwx----- 1 root root 64 Oct  7 03:12 0 -> /dev/pts/0
lrwx----- 1 root root 64 Oct  7 03:12 1 -> /dev/pts/0
lrwx----- 1 root root 64 Oct  7 03:12 2 -> /dev/pts/0
lr-x----- 1 root root 64 Oct  7 03:12 3 -
> /root/fruit01.txt
```

```
lr-x----- 1 root root 64 Oct 7 03:12 4 -  
> /root/fruit01.txt  
lr-x----- 1 root root 64 Oct 7 03:12 5 -> /proc/2069/fd  
[root@localhost ~]# cat /dev/fd/4  
banana  
apple  
carrot
```

17.2.3 Here Document

Here Document 又称此处文档，用于在命令或脚本中按行输入文本。Here Document 的格式为 `<<delimiter`，其中 `delimiter` 是一个用于标注的“分隔符”，该分隔符后所有的输入都被当作是输入的文本，直到出现下一个分隔符为止。

以17.2.1节“标准输入重定向”中用到的 `sort` 命令为例，如果在输入的过程中需要使用 `Ctrl+D` 组合键发送输入完成的信号，这在交互的环境中是可以的，但由于在脚本中无法使用组合键，因此要终止输入就需要用到 Here Document。同样的输入内容演示如下：

```
[root@localhost ~]# sort << END
> banana
> apple
> carrot
> END
apple
banana
carrot
```

再以 `cat` 命令为例，要将输入的内容保存到 `HelloWorld02.txt` 中，示例如下：

```
[root@localhost ~]# cat >> HelloWorld02.txt << END
> Hello
> World
> END
[root@localhost ~]# cat HelloWorld02.txt
Hello
World
```

第18章 脚本范例

18.1 批量添加用户脚本

用户管理是Linux系统维护的工作之一，其中涉及用户添加、删除等简单操作。但是如果需要一次性添加几十个还甚至上百个用户，这时候再简单地重复性使用`useradd`进行添加就非常低效了。现在来分析一下要完成这个工作的场景和必要的实现方式。

首先，需要一个包含所有需添加用户的用户名和密码的文本文件，该文件以行为单位，每行是一条用户信息，用户名和密码之间使用特定的分隔符分开，可以是空格、Tab键或冒号等（不同的分隔符只是脚本处理时的方法不同，没有本质区别）。为了让编写该脚本的过程遇到更多的问题，这里特意选择使用空格作为分隔符，根据这个方案写出的文本文档如下所示：

```
[root@localhost ~]# cat addusers.txt
username001 password001
username002 password002
username003 password003
username004 password004
username005 password005
username006 password006
```

现在我们面临的是如何根据这个文件添加用户了。想一想，如果是让你根据这个文本来手工添加用户，你会如何操作？大概首先想到的是从第一行开始操作，第一行的用户名是`username001`，密码是`password001`，然后使用`useradd username001`增加用户，再使用`passwd username001`给用户设置密码，以此类推，直到完成最后一个用户的添加操作——注意

这是一个循环，所以需要使⤵用循环来让系统做这件事情。问题又来了，Shell中的循环有很多种，有for循环、while循环、until循环，应该使用哪一种循环呢？

使用for循环和while循环都可以较方便地按行读取，但是笔者更倾向于使用while循环，因为while循环在按行读取时有着天然的优势。先看一下使用for循环按行读取脚本的情况，以及运行结果。

```
#for
循环按行读取脚本
[root@localhost ~]# cat useradd_for01.sh
#!/bin/bash
COUNT=0
for LINES in `cat addusers.txt`
do
    echo $LINES
    let COUNT+=1
done
echo
echo "$0 looped $COUNT times"
#for
循环按行读取脚本运行结果
[root@localhost ~]# bash useradd_for01.sh
username001
password001
username002
password002
username003
password003
username004
password004
username005
password005
username006
password006
useradd_for01.sh looped 12 times
```

从上面的脚本运行输出可以看出，该脚本实际上并没有做到“按行读取”，因为它实际上循环了12次！不是应该循环6次

吗，怎么会是12次呢？这是因为addusers.txt文件中的用户名和密码是使用空格隔开的，而for循环在读取文件时，任何空白字符都可以作为其读取的分隔符，所以它其实循环了12次。

来看看while循环的按行读取脚本的情况，以及运行结果。

```
#while
循环按行读取脚本
[root@localhost ~]# cat useradd_while01.sh
#!/bin/bash
COUNT=0
while read LINES
do
    echo $LINES
    let COUNT+=1
done < addusers.txt
echo
echo "$0 looped $COUNT times"
#while
循环按行读取脚本运行结果
[root@localhost ~]# bash useradd_while01.sh
username001 password001
username002 password002
username003 password003
username004 password004
username005 password005
username006 password006
useradd_while01.sh looped 6 times
```

从脚本运行结果来看，while的按行读取确实没有问题，因为while使用的是换行符作为行标记。如果一开始我们决定使用其他符号作为分隔符（比如冒号），那for循环也能胜任该项工作，但是这样读者就不会注意到这个问题了。这也是前面笔者特意选择空格作为分隔符的原因。

接下来需要使用字符工具对每行进行加工：从每行中分拆出用户名和密码。每行空格前的部分是用户名，空格后的部分为密码。只需要使用cut命令就能很简单地分拆开来了，把

useradd_while01.sh升级成下面的内容，并运行。查看结果，确认脚本正确地截取了需要的信息。

```
[root@localhost ~]# cat useradd_while02.sh
#!/bin/bash
while read LINES
do
    USERNAME=`echo $LINES | cut -f1 -d' '`
    PASSWORD=`echo $LINES | cut -f2 -d' '`
    #
    测试打印截取结果
    echo -n "USERNAME:$USERNAME PASSWORD:$PASSWORD"
    echo
done < addusers.txt
#
脚本成功截取了用户名和密码
[root@localhost ~]# bash useradd_while02.sh
USERNAME:username001 PASSWORD:password001
USERNAME:username002 PASSWORD:password002
USERNAME:username003 PASSWORD:password003
USERNAME:username004 PASSWORD:password004
USERNAME:username005 PASSWORD:password005
USERNAME:username006 PASSWORD:password006
```

现在既然可以成功地截取到每行的用户名、密码，那么事情就变得简单了：在新增用户时，只需要先使用useradd USERNAME命令、然后使用passwd USERNAME命令就可以了。

不过这里有个问题，还记得passwd命令是怎么运行的吗——它要求管理员手工输入密码。有没有办法能直接将密码当作一个参数传给passwd命令呢？答案是肯定的：运行man passwd。查看该命令的用法可以发现，通过使用--stdin参数，可以使用管道将密码传给passwd命令。

```
--stdin
This option is used to indicate that passwd should read th
```

from standard input, which can be a pipe.

至此该脚本的基本框架就分析完了，最后将脚本修改为如下内容，运行并观察结果。

```
[root@localhost ~]# cat useradd_while03.sh
#!/bin/bash
while read LINES
do
    USERNAME=`echo $LINES | cut -f1 -d' '`
    PASSWORD=`echo $LINES | cut -f2 -d' '`
    useradd $USERNAME
    echo $PASSWORD | passwd --stdin $USERNAME
done < addusers.txt
```

#

脚本运行结果

```
[root@localhost ~]# bash useradd_while03.sh
Changing password for user username001.
passwd: all authentication tokens updated successfully.
Changing password for user username002.
passwd: all authentication tokens updated successfully.
Changing password for user username003.
passwd: all authentication tokens updated successfully.
Changing password for user username004.
passwd: all authentication tokens updated successfully.
Changing password for user username005.
passwd: all authentication tokens updated successfully.
Changing password for user username006.
passwd: all authentication tokens updated successfully.
```

虽然说这个脚本现在已经“可以工作”了，但还不是那么完美。如果你再运行一遍该脚本，会发现新增用户是失败的（因为之前运行过一次后，在运行时用户已经存在），但是却又成功地“修改”了用户的密码，这是很危险的。所以需要在脚本中增加用户判断的环节：如果用户已存在，则跳过不做任何修改，否则增加用户。另外，在可能的情况下，所有非Shell内建命令都建议使用全路径，以避免由于环境变量的问题造成的 command not found。最后，脚本主体要尽量少使用常量，所以

需要在脚本的开头多定义变量。按上面的要求完成脚本修改，最终形式如下：

```
[root@localhost ~]# cat useradd_while04.sh
#!/bin/bash
USERS_INFO=/root/addusers.txt
USERADD=/usr/sbin/useradd
PASSWD=/usr/bin/passwd
CUT=/bin/cut
while read LINES
do
    USERNAME=`echo $LINES | $CUT -f1 -d' '`
    PASSWORD=`echo $LINES | $CUT -f2 -d' '`
    $USERADD $USERNAME
    if [ $? -ne 0 ]; then
        echo "$USERNAME exists, skip set password"
    else
        echo $PASSWORD | $PASSWD --stdin $USERNAME
    fi
done < $USERS_INFO
```

读者可以在此基础上增加更多的功能和判断使其变得更为完美。

18.2 检测服务器存活

检测服务器存活是日常运维工作中很重要也是很基础的服务器监控任务，最简单的方法是使用ping命令检测。本节将利用ping和简单的HTML语言创建服务器存活监控脚本，监控结果可通过网页展示，方便监控人员查看。

对于该脚本，需要设计成节点可配置的，也就是脚本和数据分开。数据部分使用一个文件列表，记录所有需要监控的主机的IP地址，脚本定时运行并生成HTML页面。为了能访问到生成的页面，应首先使用yum安装Apache服务，并确保服务启动。示例如下：

```
[root@localhost ~]# cat server_alive01.sh
#!/bin/bash
TIMESTAMP=`date +%Y%m%d%H%M%S`
CURRENT_HTML=/var/www/html/${TIMESTAMP}.html
CURRENT_INDEX=/var/www/html/index.html
LN=/bin/ln
RM=/bin/rm
SERVER_LIST=server_list
cat <<EOF > $CURRENT_HTML
<html>
<head>
<title>Server Alive Monitor</title>
</head>
<body>
<table width="50%" border="1" cellpadding="1" cellspacing="0"
<caption><h2>Server Alive Status</h2></caption>
<tr><th>Server IP</th><th>Server Status</th></tr>
EOF
while read SERVERS
do
#
如果ping
的结果返回0
则状态是OK
的，同时显示字体的颜色为blue
#
```

```

如果ping
的结果返回非0
则状态是FALSE
的，同时显示字体的颜色为red
ping $SERVERS -c 3
if [ $? -eq 0 ]; then
    STATUS=OK
    COLOR=blue
else
    STATUS=FALSE
    COLOR=red
fi
echo                                "<tr><td>$SERVERS</td><td>
<font color=$COLOR>$STATUS</font></td></tr>" >>
$CURRENT_HTML
done < $SERVER_LIST
cat <<EOF >> $CURRENT_HTML
</table>
</body>
</html>
EOF
#
将web
根目录中的主文件连接到新生成的页面
$LN -sf $CURRENT_HTML $CURRENT_INDEX

```

在该脚本相同目录中创建server_list文件，按行写入需要监控的服务器的IP地址。这里写了两条IP记录，其中192.168.61.131可用，而192.168.61.132是一个当前不存在的IP。

运行脚本后，在Apache的文件根目录中会生成新的index.html连接文件，可以将该脚本加入系统计划任务中（cron）定时运行（比如每分钟运行一次），生成HTML文件后，使用浏览器观察检测结果，如图18-1所示。

```

#
添加系统crontab
任务
[root@localhost ~]# crontab -l
*/1 * * * * /root/server_alive01.sh

```

```
#
几分钟后观察apache
根文件目录中确实出现了生成的页面
[root@localhost html]# ll
total 8
-rw-r--r-- 1 root root 406 Jun 11 10:33 20130611103301.html
-rw-r--r-- 1 root root 310 Jun 11 10:34 20130611103401.html
lrwxrwxrwx 1 root root 33 Jun 11 10:33 index.html -
> /var/www/html/20130611103301.html
```

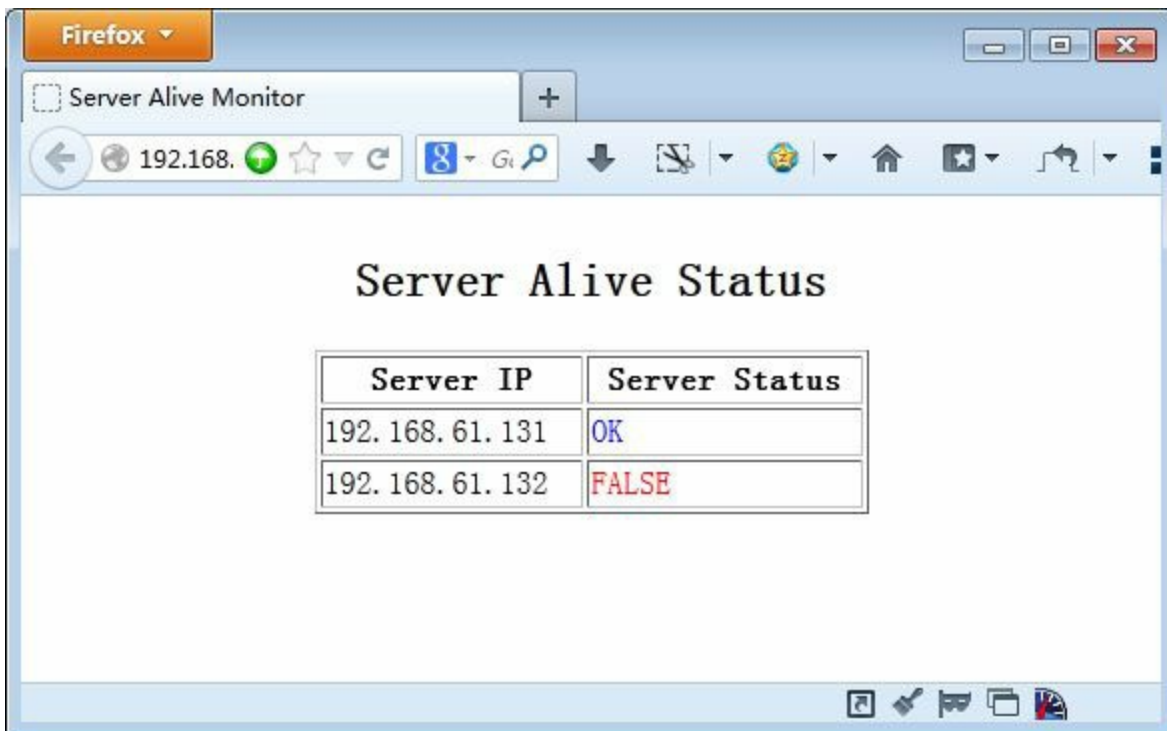


图18-1 用浏览器访问检测页面

18.3 使用expect实现自动化输入

从字面上就能大概猜出expect的用途，即“期待”系统的输出，且对应地发送输入作为响应。在man文件中是这么介绍expect的：expect是一种能够按照脚本内容设定的方式和交互程序进行“对话”的程序。

由于在Linux中的一些命令不太适合于脚本化的自动运行，比如fdisk、telnet、ftp连接下载等，所以就必须使用expect来解决这些场景下的自动化运行问题。默认情况下系统不会安装expect工具，所以需要先安装再使用，最简单的方法是使用yum安装。

本节将使用expect脚本实现ftp自动登录并下载文件。为演示脚本运行效果，需准备两台虚拟机，按照以下步骤搭建实验环境并运行之。

服务器A：配置为ftp服务器（假设IP为192.168.61.131），如下所示：

```
#
安装vsftpd
[root@localhost ~]# yum install vsftpd
#
安装完成后启动vsftpd
服务
[root@localhost ~]# service vsftpd start
Starting vsftpd for vsftpd: [ OK ]
#
在/var/ftp
目录中创建TestDownload
文件
[root@localhost ~]# touch /var/ftp/TestDownload
```

服务器B：创建expect脚本并运行，如下所示。

```
#
安装expect
[root@localhost ~]# yum install expect
#
编辑如下的expect
脚本expect_ftp_auto.exp
[root@localhost ~]# cat expect_ftp_auto.exp
#!/usr/bin/expect -f
set ip [lindex $argv 0 ]          #
脚本的第一个参数，远程主机的IP
地址
set file [lindex $argv 1 ]        #
脚本的第二个参数，指定下载的文件名
set timeout 10
spawn ftp $ip                      #
运行ftp $ip
命令
expect "Name*"                    #
如果出现Name
字符
send "anonymous\r"                #
则输入anonymous
并回车
expect "Password:*"               #
如果出现Password
字符
send "\r"                          #
则仅输入回车
expect "ftp>*"                     #
如果出现ftp>
字符
send "get $file\r"                 #
则发送get $file
命令
expect {
"*Failed*" { send_user " Download failed\r";send "quit\r" }
#
如果返回的字符串中含有Failed
则说明下载失败
"*send*" { send_user " Download ok\r";send "quit\r"}
#
如果返回的字符串中含有send
则说明下载成功
}
expect eof
```



```
#
给脚本expect_ftp_auto.exp
加上可执行权限
[root@localhost ~]# chmod +x expect_ftp_auto.exp
#
运行脚本，连接192.168.61.131
的ftp
服务器并下载TestDownload
文件
[root@localhost ~]# ./expect_ftp_auto.exp 192.168.61.131 Test
spawn ftp 192.168.61.131
Connected to 192.168.61.131.
220 (vsFTPd 2.0.5)
530 Please login with USER and PASS.
530 Please login with USER and PASS.
KERBEROS_V4 rejected as an authentication type
Name (192.168.61.131:root): anonymous
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> get TestDownload
local: TestDownload remote: TestDownload
227 Entering Passive Mode (192,168,61,131,181,165)
150 Opening BINARY mode data connection for TestDownload (0 b
226 File send OK.
quitnload ok
ftp> 221 Goodbye.
#
下载一个不存在的文件TestDown
，观察脚本的运行输出
[root@localhost ~]# ./expect_ftp_auto.exp 192.168.61.131 Test
spawn ftp 192.168.61.131
Connected to 192.168.61.131.
220 (vsFTPd 2.0.5)
530 Please login with USER and PASS.
530 Please login with USER and PASS.
KERBEROS_V4 rejected as an authentication type
Name (192.168.61.131:root): anonymous
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> get TestDown
local: TestDown remote: TestDown
227 Entering Passive Mode (192,168,61,131,106,149)
```

550 Failed to open file.
quitnload failed
ftp> 221 Goodbye.

18.4 自动登录ftp备份

ftp是很常见的用于存取文件的应用，它也用于日常备份。这种周期性的工作无疑需要通过自动化脚本来完成，除了上一节中使用的expect方法外，还可以利用重定向技巧自动登录ftp服务器。本节将沿用上节的实验环境，但会做一些修改。

在下面的示例中，修改ftp服务器的配置文件/etc/vsftpd/vsftpd.conf，将anon_upload_enable=YES前面的注释符去掉，即允许匿名用户上传文件；在下一行中增加anon_other_write_enable=YES，即允许匿名用户覆盖同名文件，并重启vsftpd服务使配置生效。

```
[root@localhost ~]# service vsftpd restart
Shutting down vsftpd: [ OK ]
Starting vsftpd for vsftpd: [ OK ]
```

创建上传目录并赋予合理的权限，命令如下：

```
[root@localhost ~]# mkdir /var/ftp/upload
[root@localhost ~]# chown ftp:root /var/ftp/upload
#
下面的脚本运行后，使用如下命令确认文件经由脚本上传成功
#[root@localhost ~]# ll /var/ftp/upload/TestUpload
#-rw-----
- 1 ftp ftp 0 Jun  6 23:01 /var/ftp/upload/TestUpload
```

在另一台服务器上创建ftp自动上传下载脚本，并运行。

```
[root@localhost ~]# touch TestUpload #
在本地创建TestUpload
用于脚本上传
```

```

[root@localhost ~]# cat autoftp01.sh
#!/bin/bash
GET_FILENAME="TestDownload"
PUT_GET_FILENAME="TestUpload"
SERVER_IP="192.168.61.131"
USER="anonymous"
PASS=""
FTP=/usr/bin/ftp
$FTP -n $SERVER_IP << EOF #
使用-n
参数关闭ftp
的自动登录模式，改为使用Here Document
quote USER $USER #quote
为ftp
的命令，可传入账号和密码
quote PASS $PASS
Binary #
指定传输方式是二进制
get $FILENAME #get
命令用于下载文件
cd upload #
进入upload
目录
put $PUT_FILENAME #put
命令用于上传文件
EOF
#
运行脚本
[root@localhost ~]# bash autoftp01.sh #
脚本运行没有任何输出
[root@localhost ~]# ll TestDownload #
本地成功下载了这个文件
-rw-r--r-- 1 root root 0 Jun 28 01:40 TestDownload

```

以上将ftp的用户名和密码以明文的方式写在了脚本中，这可能是个安全隐患。因此，可以利用ftp的自动登录模式，将用户名和密码分离出去。默认情况下，直接使用ftp命令并回车就会进入ftp的自动登录模式，此时ftp命令会读取用户家目录下的.netrc文件，默认情况下这个文件是不存在的，所以ftp会提示输入用户名和密码信息。示例如下：

```

[root@localhost ~]# ftp 192.168.61.131

```

```
Connected to 192.168.61.131.
220 (vsFTPd 2.0.5)
530 Please login with USER and PASS.
530 Please login with USER and PASS.
KERBEROS_V4 rejected as an authentication type
Name (192.168.61.131:root):
```

.netrc文件的格式如下所示:

```
machine IP login <username> password <password>
#
按照该格式创建.netc
文件
[root@localhost ~]# cat .netrc
default login anonymous password '\r'  #
空密码使用特殊字符'\r'
表示
```

再次运行ftp命令时，发现已经可以自动登录了。

```
[root@localhost ~]# ftp 192.168.61.131
Connected to 192.168.61.131.
220 (vsFTPd 2.0.5)
530 Please login with USER and PASS.
530 Please login with USER and PASS.
KERBEROS_V4 rejected as an authentication type
331 Please specify the password.
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
```

最后利用.netrc自动登录，将autoftp01.sh脚本改为autoftp02.sh，可完成同样的功能，如下所示:

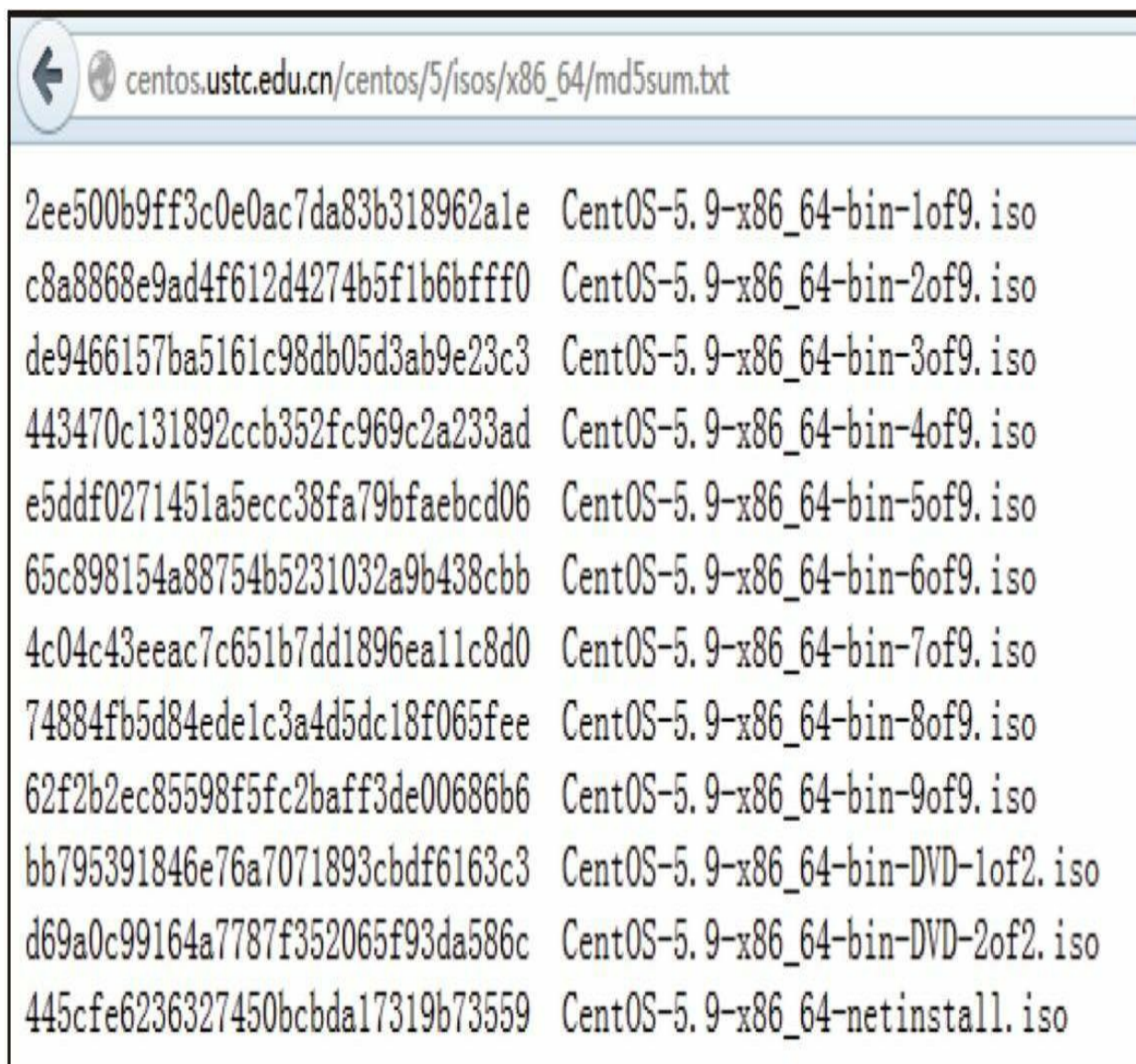
```
[root@localhost ~]# cat autoftp01.sh
#!/bin/bash
GET_FILENAME="TestDownload"
```

```
PUT_FILENAME="TestUpload"
SERVER_IP="192.168.61.131"
FTP=/usr/bin/ftp
$FTP $SERVER_IP << EOF
binary
get $GET_FILENAME
cd upload
put $PUT_FILENAME
EOF
```

18.5 文件安全检测脚本

到这里，相信Linux下“一切皆文件”的理念已经深入人心了，所以从很大程度上来说文件安全是系统安全很重要的部分。服务器在完成初始化安装时，所有文件都是从安装介质中获得的，不存在任何问题。但是随着服务器进入生成环境、各类应用上线、系统和软件漏洞、管理员介入管理等各方面因素的产生，主机的安全就成了必须考虑的问题。

大家很可能在平时下载软件时已经注意到，有些软件在下载链接周围会包含一串含有数字和字母的字符串，又叫MD5值，其实这是一种文件安全机制：用户下载文件后，使用MD5算法对该文件进行计算，并将该值和网站公布的值进行比对，若结果一致则说明文件和源文件是一致的，可以放心使用。CentOS在其下载镜像中也提供了原始MD5值，以方便用户在下载后进行比对，如图18-2所示。



2ee500b9ff3c0e0ac7da83b318962a1e	CentOS-5.9-x86_64-bin-1of9.iso
c8a8868e9ad4f612d4274b5f1b6bfff0	CentOS-5.9-x86_64-bin-2of9.iso
de9466157ba5161c98db05d3ab9e23c3	CentOS-5.9-x86_64-bin-3of9.iso
443470c131892ccb352fc969c2a233ad	CentOS-5.9-x86_64-bin-4of9.iso
e5ddf0271451a5ecc38fa79bfaebcd06	CentOS-5.9-x86_64-bin-5of9.iso
65c898154a88754b5231032a9b438cbb	CentOS-5.9-x86_64-bin-6of9.iso
4c04c43eeac7c651b7dd1896ea11c8d0	CentOS-5.9-x86_64-bin-7of9.iso
74884fb5d84edc1c3a4d5dc18f065fee	CentOS-5.9-x86_64-bin-8of9.iso
62f2b2ec85598f5fc2baff3de00686b6	CentOS-5.9-x86_64-bin-9of9.iso
bb795391846e76a7071893cbdf6163c3	CentOS-5.9-x86_64-bin-DVD-1of2.iso
d69a0c99164a7787f352065f93da586c	CentOS-5.9-x86_64-bin-DVD-2of2.iso
445cfe6236327450bcbda17319b73559	CentOS-5.9-x86_64-netinstall.iso

图18-2 CentOS镜像文件的MD5值

这种计算MD5值的算法叫做“哈希算法（md5 checksum）”，理论上如果文件没有经过修改，不管移动到哪里，对其进行哈希计算得到的值都应该完全一样。虽然MD5算法存在极小几率的碰撞（即不同的文件也可能算出一样的MD5值），但用于日常工作足以胜任。下面是计算某个文件MD5值的示例，并且可验证文件是否被修改。

```
#
计算/etc/passwd
的MD5
值，并将计算结果保存到文件中
[root@localhost ~]# md5sum /etc/passwd > passwd.md5
#
验证该文件是否被修改，如果没有则打印OK
[root@localhost ~]# md5sum -c passwd.md5
/etc/passwd: OK
#
添加用户后，再次验证，此时MD5
值发生了改变，验证不通过
[root@localhost ~]# md5sum -c passwd.md5
/etc/passwd: FAILED
md5sum: WARNING: 1 of 1 computed checksum did NOT match
#
也可以用--status
参数使命令不产生文字输出
[root@localhost ~]# md5sum -c --status passwd.md5
[root@localhost ~]# echo $?
1
```

下面的脚本用于找出指定目录列表内的所有普通文件，并进行MD5值计算。值得提醒的是，读者有必要在脚本中添加远程文件复制的功能，将生成的MD5文件保存到远程备份服务器，或参考上一小节自动登录到ftp服务器上备份，否则一旦发生该主机被入侵的情况，入侵者不但可以修改文件内容，也可以对该MD5文件进行修改，这样就无从对文件的一致性进行检查了。而如果将该MD5文件进行远程备份，则可以降低风险。

```
[root@localhost ~]# cat md5check.sh
#!/bin/bash
DIRS="/bin /usr/bin"
FIND=/usr/bin/find
MD5SUM=/usr/bin/md5sum
MD5FILE=md5sum.md5
$FIND $DIRS -type f | while read line
do $MD5SUM $line >> $MD5FILE
done
```

18.6 ssh自动登录备份

利用ssh可实现远程登录主机并执行命令的功能，还可以在远程主机上进行资料备份。其使用格式如下：

```
ssh user@remote_ip "COMMAND"
#ssh
远程登录到192.168.61.131
执行echo
命令
[root@localhost ~]# ssh root@192.168.61.131 "echo $HOSTNAME"
The authenticity of host '192.168.61.131 (192.168.61.131)' ca
RSA key fingerprint is 30:72:cf:dc:2d:74:e4:c2:ed:72:a9:22:e9
Are you sure you want to continue connecting (yes/no)? Yes #
输入yes
确认连接
Warning: Permanently added '192.168.61.131' (RSA) to the list
root@192.168.61.131's password: #
输入192.168.61.131
的密码
Localhost.localdomain #
命令输出结果
```

从上面的演示可以看出，使用ssh确实可以实现远程执行命令，只需要将对应的COMMAND命令改为备份命令即可。不过细心的读者可能会发现了一个问题：命令执行的过程中需要人为地输入远程主机的密码，如果是第一次ssh连接还需要输入yes确认连接。如果只是一次输入还没有什么影响，但是如果有上百台目标主机，那就需要重复上百次，这无疑是不能接受的。所以首先需要实现远程执行命令无须人工输入密码的问题，可以通过以下3个步骤实现。

第一步，使用ssh-keygen创建公钥和私钥。

```
[root@localhost ~]# ssh-keygen
```

```

Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa): #
回车
Enter passphrase (empty for no passphrase): #
回车
Enter same passphrase again: #
回车
Your identification has been saved in /root/.ssh/id_rsa.
Your public key has been saved in /root/.ssh/id_rsa.pub.
The key fingerprint is:
3d:b9:15:f9:9e:c4:09:a8:71:75:12:21:48:17:e4:c7 root@localhos
You have new mail in /var/spool/mail/root
#
创建完成后会在/root/.ssh/
目录下生成两个文件，分别是公钥和私钥
[root@localhost ~]# cd .ssh/
[root@localhost .ssh]# ll
total 12
-rw----- 1 root root 1675 Jun 26 01:42 id_rsa          #
私钥
-rw-r--r-- 1 root root  408 Jun 26 01:42 id_rsa.pub       #
公钥

```

第二步，使用ssh-copy-id复制公钥到远程主机。

```

[root@localhost ~]# ssh-copy-id -
i .ssh/id_rsa.pub root@192.168.61.131
root@192.168.61.131's password: #
输入192.168.61.131
的密码
Now try logging into the machine, with "ssh 'root@192.168.61.
.ssh/authorized_keys
to make sure we haven't added extra keys that you weren't exp
#
复制过程实质上是将本地/root/.ssh/id_rsa.pub
中的内容复制追加到远程主机的
/root/.ssh/authorized_keys
文件中，如果远程主机不存在这个文件，则自动创建该文件
（默认该文件是不存在的）。下面是本地/root/.ssh/id_rsa.pub
文件的内容
[root@localhost ~]# cat .ssh/id_rsa.pub
ssh-
rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAtjTSdUWvkInSJUkPKeK+Xhl07cpE7
U0G8K0qsMUm0G7QwZ3RDPTtHOM9en1py/pTLU9X+z0PKHkUpip+Qb/CPyh7CM6

```

```
Dw5U4RJwhMdXetRY/YsxmDL3/dcFYw+47adyALFH3f3ZYHvBP/V3e3E9a5aTu
Yy6qAueGegqZuh0Ap6aEEyL3L00VxDIPEJS6vhJwAHY7mkldw0EdPZg4quHOM
NnaaEafanTuZaB21D6Y8cf1mn9U33VG0E3DWb+jM2qA3gNP1q4bpP20o7fBH2
WoFrBhC7cS0eapJk0cEv1050Lz05tQ== root@localhost.localdomain
#
```

下面是通过ssh-copy-id

命令复制到远程主机192.168.61.131

后，远程主机

/root/.ssh/authorized_keys

文件的内容，可以看到文件和之前生成的id_rsa.pub

内容是一致的

```
[root@localhost ~]# cat .ssh/authorized_keys ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAQEAtjTSdUWvkInSJUKPKek+Xhl07cpE75LN5
U0G8K0qsMUm0G7QwZ3RDPtHOM9en1py/pTLU9X+z0PKHkUpip+Qb/CPyh7CM6
Dw5U4RJwhMdXetRY/YsxmDL3/dcFYw+47adyALFH3f3ZYHvBP/V3e3E9a5aTu
Yy6qAueGegqZuh0Ap6aEEyL3L00VxDIPEJS6vhJwAHY7mkldw0EdPZg4quHOM
NnaaEafanTuZaB21D6Y8cf1mn9U33VG0E3DWb+jM2qA3gNP1q4bpP20o7fBH2
WoFrBhC7cS0eapJk0cEv1050Lz05tQ== root@localhost.localdomain
```

第三步，尝试远程登录。

```
[root@localhost ~]# ssh root@192.168.61.131
Last login: Tue Jun  4 20:35:04 2013 from 192.168.61.1 #
发现不需要输入密码即可进入
```

上面的方法只是解决了远程登录时输入密码的问题，但是如果几十上百台主机，管理员要一台一台地ssh-copy-id也还是一件非常烦琐的事情。借助于expect工具，可以将整个过程变得自动化。

该脚本接受两个参数，第一个参数是远程主机的密码，第二个参数是远程主机的IP。该脚本执行过程中将会模拟整个ssh-copy-id行为：如果系统提示包含“(yes/no)”，则发送yes；如果系统提示包含“Password:”则发送密码。

```
[root@localhost ~]# cat expect_ssh_copy_id.sh
#!/bin/bash
```

```

PASS=$1
IP=$2
auto_ssh_copy_id () {
    expect -c "set timeout -1;
spawn /usr/bin/ssh-copy-id -
i /root/.ssh/id_rsa.pub root@$2;
    expect {
        *(yes/no)* {send -- yes\r;exp_continue;}
        *Password:* {send -- $1\r;exp_continue;}
        eof {exit 0;}
    }";
}
auto_ssh_copy_id $PASS $IP
#
使用该脚本进行自动化地执行ssh-copy-id
[root@localhost ~]# bash expect.sh 111111 192.168.61.131
spawn /usr/bin/ssh-copy-id -
i /root/.ssh/id_rsa.pub root@192.168.61.131
The authenticity of host '192.168.61.131 (192.168.61.131)' ca
RSA key fingerprint is 30:72:cf:dc:2d:74:e4:c2:ed:72:a9:22:e9
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.61.131' (RSA) to the list
root@192.168.61.131's password:
Now try logging into the machine, with "ssh 'root@192.168.61.
.ssh/authorized_keys
to make sure we haven't added extra keys that you weren't exp

```

借助这个脚本可以大规模地自动完成ssh-copy-id的操作，需要准备的就是一个文本文件，该文件中每一行为一个服务器信息，包含IP和密码。如果还不明白的话，那应该再次学习一下18.1节中的“批量添加用户脚本”——按行读取是该脚本的重点之一。

18.7 使用rsync备份

Remote Synchronize简称rsync，这是一款可以远程同步文件的软件，同步过程采用rsync加密算法保证了文件安全，并且同步的文件可保持原文件的属性（比如权限、时间等）不变。

首先准备两台服务器作为实验环境，按照以下步骤搭建实验环境。

服务器A：配置为rsync服务器（假设IP为192.168.61.130）。

```
#
安装rsync
软件
[root@localhost ~]# yum install rsync
#
安装xinetd
[root@localhost ~]# yum yum install xinetd
#
配置xinetd
、rsync
开机自启动
[root@localhost ~]# chkconfig xinetd on
[root@localhost ~]# chkconfig rsync on
#
手工创建rsync
的配置文件/etc/rsyncd.conf
[root@localhost ~]# cat /etc/rsyncd.conf
uid = root
gid = root
use chroot = no
max connections = 10
strict mode = yes
port = 873
[backup]
path = /root
comment = Root Dir
```

```
ignore errors
read only = yes
list = no
auth users = john
secret file = /etc/rsync.sec
#
配置密码文件/etc/rsync.sec
并授予600
权限
[root@localhost ~]# cat /etc/rsync.sec
john:wang001
[root@localhost ~]# chmod 600 /etc/rsync.sec
#
重启xinetd
以激活rsync
[root@localhost ~]# service xinetd restart
#
确认rsync
已经运行
[root@localhost ~]# netstat -a | grep rsync
tcp          0          0 *:rsync      *:*
```

服务器B: 配置为rsync客户端（假设IP为192.168.61.131）。

```
#
安装rsync
软件
[root@localhost ~]# yum install rsync
#
配置rsync
客户端密码文件并授予600
权限
[root@localhost ~]# cat /etc/rsync.sec
wang001
[root@localhost ~]# chmod 600 /etc/rsync.sec
#
创建目录/root/rsync_dir
, 用于同步服务器A[backup]
模块中的文件
[root@localhost ~]# mkdir rsync_dir
#
测试同步, 把服务器A/root
目录中的全部文件同步到本地/root/rsync_dir
```

目录

```
[root@localhost ~]# rsync -vzrtopg --progress --  
delete john@192.168.61.130::backup  
/root/rsync_dir --password-file=/etc/rsync.sec
```

最后脚本化以上rsync过程，增加服务器是否存活的判断，并增加日志使rsync过程更加清晰，这样便于在运行出错时排查问题。

```
#!/bin/bash  
RSYNC_SERVER=192.168.61.130  
RSYNC_USER=john  
RSYNC_MODULE=backup  
RSYNC_PASS=/etc/rsync.sec  
RSYNC_LOG=/var/run/rsync.log  
LOCAL_DIR=/root/rsync_dir  
RSYNC=/usr/bin/rsync  
PING=/bin/ping  
run_rsync() {  
    echo "Starting Rsync at `date`" | tee -a $RSYNC_LOG  
    $RSYNC -vzrtopg --progress --  
delete $RSYNC_USER@$RSYNC_SERVER::$RSYNC_  
MODULE $LOCAL_DIR --password-file=$RSYNC_PASS  
    echo "Rsync Finished at `date`" | tee -a $RSYNC_LOG  
}  
test_alive() {  
    $PING $RSYNC_SERVER -c 3 -w 3  
    if [ $? -ne 0 ]; then  
        echo "Server down at `date`" >> $RSYNC_LOG  
        exit 1  
    fi  
}  
test_alive > /dev/null 2>&1  
run_rsync
```

18.8 使用netcat备份

netcat被誉为网络工具中的“瑞士军刀”，体积虽小但是功能强大。netcat最简单的功能是由于端口扫描，下面的示例将针对指定IP地址的20~25端口进行扫描，可以看到21（ftp）、22（ssh）端口是打开的。

```
[root@localhost ~]# nc -z -v -n 192.168.61.131 20-25
nc: connect to 192.168.61.131 port 20 (tcp) failed: Connection refused
Connection to 192.168.61.131 21 port [tcp/*] succeeded!
Connection to 192.168.61.131 22 port [tcp/*] succeeded!
nc: connect to 192.168.61.131 port 23 (tcp) failed: Connection refused
nc: connect to 192.168.61.131 port 24 (tcp) failed: Connection refused
nc: connect to 192.168.61.131 port 25 (tcp) failed: Connection refused
```

除了端口扫描外，netcat还有通过网络传输文件的功能，它能够通过TCP或UDP协议在网络中读写数据。简单来说，netcat所做的就是在两台服务器之间建立连接并交流数据，这种功能很容易让我们想到可以利用netcat通过网络备份数据。虽然在Linux下有很多传输文件的方法，比如SCP、FTP、SMB、NFS等，但是netcat在和其他小工具结合后，可拥有更为灵活的功能。比如与tar命令结合，可以在压缩解压的同时传输文件；与mccrypt结合，可以一边加密解密一边传输文件；与mplayer命令结合，可以在接收数据的同时播放视频；与dd结合，可以在dump文件的同时保存到远端，等等。

在备份文件较大的情况下，典型的备份过程是先使用工具压缩打包源文件，该步骤耗时Time1；完成此操作后，为了数据安全，会将生成的压缩文件同时复制到远端某个服务器集中存放，该步骤耗时Time2，则整个过程耗时为 $X=Time1+Time2$ 。如果需要备份的文件特别大，网络质量也不是很好，整体耗时X会显著增加。

来做一个简单的计算：笔者测试了运行在VMware上的虚拟机的磁盘I/O速度（宿主机是某品牌的SSD磁盘），在以8KB为单次I/O的同时读写速度为40MB/s（注意：在实际生成环境下，由于服务器还有其他业务负载，而且SSD硬盘目前成本较高，在服务器端还没有形成大规模部署，所以数据不会这么乐观），30秒内完成了1.2GB的数据读写，粗略计算完成120GB的数据打包操作需耗时50分钟。按照文件压缩比为25%计算，生成的压缩文件为30GB，通过百兆局域网复制耗时约为41分钟（注意：广域网的网络质量会明显比百兆局域网的网络质量差），所以整体耗时91分钟。同样的备份场景，如果使用tar命令，在打包的同时通过管道给netcat，经由netcat传输到远端备份服务器。由于打包和传输过程从串行变为了并行，所以整体耗时将会降低到50分钟，时间占比降低了45%，效果非常显著。示例如下：

```
#
基于VMware
上一台虚拟机的磁盘I/O
性能（宿主机使用某品牌SSD
硬盘）
[root@localhost ~]# dd if=/dev/sda of=/tmp/RW.img bs=8k
151935+0 records in
151935+0 records out
1244651520 bytes (1.2 GB) copied, 30.8994 seconds, 40.3 MB/s
#
国内某云主机的磁盘I/O
性能
[root@localhost ~]# dd if=/dev/sda of=/tmp/RW.img bs=8k
98663+0 records in
98663+0 records out
808247296 bytes (808 MB) copied, 31.5515 seconds, 25.6 MB/s
```

现在来了解一下netcat实现文件传输的方法。首先，依次在两台服务器上安装nc工具。

```
[root@localhost ~]# yum install nc
```

然后，使用nc命令在文件接收服务器（又称服务端）绑定一个端口，该命令在服务器B上执行。

```
#  
指定绑定本地端口1234  
，其实就是创建了一个socket  
端口  
#  
读者可以任意使用一个未被占用的端口，建议大于1024  
#  
这条命令的含义是：监听本地1234  
端口，并将在该端口上收到的数据保存到file.rec  
文件中  
[root@localhost ~]# nc -l 1234 > file.rec &
```

最后，在发送文件的服务器上（又称客户端，此处是服务器A）向这个网络socket中发送数据。可采取下面任意一种方式：

```
#cat  
本地文件install.log  
，输出内容通过管道重定向到网络socket  
中  
[root@localhost ~]# cat install.log | nc 192.168.61.131 1234  
#  
下面的命令和上面的等价  
#[root@localhost ~]# nc 192.168.61.131 1234 < install.log
```

读者到服务器B上查看一下file.rec文件，会发现，其内容和服务器A上的install.log是完全一致的。当然了，file.rec就是install.log的一个网络拷贝！

实际上，不仅是文本文件，其他类型的文件甚至是磁盘分区都可以通过这种方式传输。下面的例子演示使用netcat传送

二进制内核文件。要说明的是，一旦服务端在监听端口上完成了数据接收，就会自动关闭该端口，所以服务端每次接收数据都需要重新绑定端口。

```
#
发送服务器B
的vmlinuz-2.6.18-194.el5
文件到服务器A
（二进制类型）
#
服务器A
上运行
[root@localhost ~]# nc -l 1234 > kernel.rec &
#
服务器B
上运行
[root@localhost ~]# nc 192.168.61.131 1234 < /boot/vmlinuz-
2.6.18-194.el5
#
发送服务器B
的sda1
分区到服务器A
#
服务器A
上运行
[root@localhost ~]# nc -l 1234 | dd of=./sda1.img
208763+41 records in
208782+0 records out
106896384 bytes (107 MB) copied, 12.4768 seconds, 8.6 MB/s
#
服务器B
上运行
[root@localhost ~]# dd if=/dev/sda1 | nc 192.168.61.131 1234
208782+0 records in
208782+0 records out
106896384 bytes (107 MB) copied, 7.82997 seconds, 13.7 MB/s
```

从上面的例子可以看到，通过netcat传输文件需要服务端（接收端）和客户端（发送端）协同工作，而且必须要服务端先打开监听端口，然后才能从客户端发送文件。所以如果是例行性地文件传送，就需要用到crontab，并且要故意设置一定的

时间差来协调这种备份。

现有两台服务器，分别是服务器A（192.168.61.130）、服务器B（192.168.61.131），需求是每天把服务器A的整个/root目录打包备份到服务器B的/root/backup目录中，并以日期和时间戳作为后缀进行归档，以方便日后查找。下面在服务器B上创建以下脚本给予执行权限，并创建crontab任务。

```
[root@localhost ~]# cat server01.sh
#!/bin/bash
NC=/usr/bin/nc
TIMESTAMP=`date +%Y%m%d%H%M%S`
PORT=1234
$NC -l $PORT > root.$TIMESTAMP.tgz
[root@localhost ~]# chmod +x server01.sh
#
创建计划任务如下，每天凌晨1
点打开端口准备接收备份数据
[root@localhost ~]# crontab -l
0 1 * * * /root/server01.sh
```

在服务器A上创建以下脚本并给予执行权限，并创建crontab任务。

```
[root@localhost ~]# cat client01.sh
#!/bin/bash
NC=/usr/bin/nc
TAR=/bin/tar
BACKUP_DIR=/root
PORT=1234
SERVER_IP=192.168.61.131
$TAR -zvcf - $BACKUP_DIR | nc $SERVER_IP $PORT
[root@localhost ~]# chmod +x client01.sh
#
创建计划任务如下，每天凌晨1
点01
分开始向服务器传送备份数据
[root@localhost ~]# crontab -l
1 1 * * * /root/client01.sh
```

建议读者先在服务端和客户端手工运行一下脚本，确保脚本本身工作正常后再使用系统计划任务执行。另外，请注意两台服务器的时间一定要正确同步。

18.9 使用iptables建立防火墙

iptables是Linux下功能强大的防火墙工具，由于它集成于Linux内核，所以效率极高。该工具在系统安装的过程中会默认安装，如果没有，可以使用RPM或yum安装。如果想自行编译安装最新的版本也很简单：下载最新的源码包，然后执行`./configure--prifix=/some/path/&&make&&make install`命令就可以了。关于软件的安装过程此处不赘述。

按照对数据包的操作类别分类，iptables可以分为4个表，按照不同的Hook点可区分为5个链。其中4个表分别是filter表（用于一般的过滤）、nat表（地址或端口映射）、mangle表（对特定数据包的修改）、raw表，这里面最常用的是filter表；5个链分别是PREROUTING链（数据包进入路由决策之前）、INPUT（路由决策为本机的数据包）、FORWARD（路由决策不是本机的数据包）、OUTPUT（由本机产生的向外发送的数据包）、POSTROUTING（发送给网卡之前的数据包），最常用的是INPUT、OUTPUT链。

关于iptables防火墙的知识足够专门用一本书来描述，本书无法面面俱到。本节旨在给读者演示一个功能完整的防火墙开发过程，读者可以根据实际需要进行修改后投入真实的生产环境。

防火墙的工作策略一般包含两种方式，第一种是仅接受允许的数据，这种策略一般是设置防火墙的默认策略为拒绝所有数据包（也就是拒绝网卡上出入的数据包），然后有针对性地放开特定的访问；第二种是只防止不允许的数据访问请求，这种策略一般是设置防火墙的默认策略为允许所有数据包，只拒绝已知的非法访问数据。从安全效果而言，前一种防火墙策略表现更为优秀，所以这里使用第一种策略来开发防火墙脚本。

首先在使用iptables之前输入以下两条命令，可以将其理解为防火墙的初始化（这里不做深入介绍）。

```
iptables -F      #  
清空所有规则  
iptables -X      #  
删除所有自定义的链
```

下面开始建立iptables防火墙规则。我们采取的规则是：默认所有的数据都丢弃，仅接收已知的数据包，所以要有针对性地打开需要的端口。下面两条命令定义默认全部丢弃数据包。

```
iptables -P INPUT DROP  
iptables -P OUTPUT DROP  
#-P  
参数的意思是policy  
， 翻译成策略  
#  
第一句的意思是：  
#  
输入(INPUT)  
的数据包默认的策略(-P)  
是丢弃(DROP)  
的  
#  
第二句的意思是：  
#  
输出(OUTPUT)  
的数据包默认的策略(-P)  
是丢弃(DROP)  
的
```

其实到这里它已经是一个有用的防火墙了，只不过没有什么意义——因为这和拔掉网线的操作没有什么不同，而且比没有防火墙更糟糕的是本地数据包都无法通信了。这种类型的防火墙需要一些基本策略来保证一些基本功能可用，所以下面的一些规则也是需要的。

```
iptables -A INPUT -p icmp --icmp-type any -j ACCEPT
#
允许icmp
包进入。如果确认不需要icmp
通信，此条可以不写
iptables -A OUTPUT -p icmp --icmp any -j ACCEPT
#
允许icmp
包出去
iptables -A INPUT -s localhost -d localhost -j ACCEPT
#
允许本地数据包出
iptables -A OUTPUT -s localhost -d localhost -j ACCEPT
#
允许本地数据包入
iptables -A INPUT -m state --state ESTABLISHED,RELATED -
j ACCEPT
#
允许已经建立和相关的数据包进入
iptables -A OUTPUT -m state --state ESTABLISHED,RELATED -
j ACCEPT
#
允许已经建立和相关的数据包出去
```

写完上面的基本策略后，现在需要考虑一些特定策略了，这和你的服务器是什么样的应用类型密切相关。如果是一台Web服务器的话，典型的需要是能访问80端口，但是就目前的策略而言是无法访问的，所以需要允许80端口的访问。命令如下：

```
iptables -A INPUT -p tcp --dport 80 -j ACCEPT
```

如果你认为这样就大功告成的话那就错了，不信你可以尝试访问一下，会发现依然打开不了Web服务器的主页（假设你设置好了Apache服务，并应用了以上的防火墙规则），为什么呢？考虑一下计算机是怎么工作的。假设你的计算机是A，服务器是B，从A发送了一个目的地址为B、目的端口是80的数据

包。服务器B收到这个数据包时发现该数据包匹配INPUT链规则，所以这个包可以正常的进入服务器B；然后服务器B在给A回包时，回包会进入本地的OUTPUT链——但是OUTPUT链默认是DROP所有包的，而且没有定义相关允许策略，回包无法出去，于是造成了整个访问的过程不完整。所以需要下面的命令：

```
iptables -A OUTPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
```

现在再试试访问Web服务器，一定是成功的。这条命令中使用了状态跟踪模块，意思是对能建立完整的连接以及为了维持该连接需要打开的其他连接所产生的数据包都是可以通过防火墙的OUTPUT链。但是如果需要允许该服务器访问其他的Web服务器，该怎么办呢？只要打开让数据出去的80端口就可以了，这需要两条命令，如下所示：

```
iptables -A OUTPUT -p tcp -m state --state NEW --dport 80 -j ACCEPT
iptables -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
```

如果此时尝试使用服务器访问某个域名，比如www.baidu.com，会发现其实还是不能访问到页面。难道是上面的规则不对吗？考虑一下使用域名访问网站需要经历什么过程。对了，域名解析。因为服务器访问该域名之前需要先解析出它的IP地址，所以防火墙必须允许域名解析的数据包出去，使用如下的命令就可以了。

```
iptables -A OUTPUT -p udp --dport 53 -j ACCEPT
```

到了举一反三的时候了，如果要访问https（默认目标端口为443）的站点，应该打开什么端口呢？这里请读者自己试一下吧。下面还列举了一些常见的需要打开的端口，读者可以参考设置。

```
#
由于管理需要ssh
到这台服务器，则需要打开22
号端口
iptables -A INPUT -p tcp -dport 22 -j ACCEPT
#
如果只允许一个固定的IP
能ssh
到该服务器的话，上面的语句需要改为
iptables -A INPUT -p tcp --dport 22 -s 192.168.1.10 -
j ACCEPT
#
可能还需要从该服务器ssh
到别的服务器
iptables -A OUTPUT -p tcp --dport 22 -j ACCEPT
```

到这里，一个简单的iptables防火墙就可以使用了，大家可以领悟一下该脚本开发过程中的各个关键点，最重要的是需要了解服务器可以正常工作时对防火墙策略的需求，以及相应的对端口放开INPUT和OUTPUT链上的策略。最后将整个过程脚本化，如下所示：

```
#!/bin/bash
#DEFINE VARIABLES
HTTP_PORT=80
SECURE_HTTP_PORT=443
SSH_PORT=22
DNS_PORT=53
ALLOWED_IP=192.168.1.10
IPTABLES=/sbin/iptables
#FLUSH IPTABLES
$IPTABLES -F
$IPTABLES -X
```

```
#DEFINE DEFAULT ACTION
$IPTABLES -P INPUT DROP
$IPTABLES -P OUTPUT DROP
#DEFINE INPUT CHAINS
$IPTABLES -A INPUT -p icmp --icmp-type any -j ACCEPT
$IPTABLES -A INPUT -s localhost -d localhost -j ACCEPT
$IPTABLES -A INPUT -m state --state ESTABLISHED,RELATED -
j ACCEPT
$IPTABLES -A INPUT -p tcp --dport $SSH_PORT -j ACCEPT
#DEFINE OUTPUT CHAINS
$IPTABLES -A OUTPUT -p icmp --icmp any -j ACCEPT
$IPTABLES -A OUTPUT -s localhost -d localhost -j ACCEPT
$IPTABLES -A OUTPUT -m state --state ESTABLISHED,RELATED -
j ACCEPT
$IPTABLES -A OUTPUT -p tcp -m state --state NEW --
dport $HTTP_PORT -j ACCEPT
$IPTABLES -A OUTPUT -p tcp --dport $SECURE_HTTP_PORT -
j ACCEPT
$IPTABLES -A OUTPUT -p udp --dport $DNS_PORT -j ACCEPT
$IPTABLES -A OUTPUT -p tcp --dport $SSH_PORT -j ACCEPT
```

18.10 自定义开机启动项的init脚本

大多数基于Linux的操作系统都使用了System-V风格的init进程管理，大量服务使用init脚本进行管理。init脚本是Linux系统用于启动系统服务的脚本，RedHat和CentOS发行版默认将这些服务启动脚本放在/etc/init.d目录中。系统在启动时将根据当前的运行级（runlevel X）确定运行在/etc/rc.d/rcX.d目录下的脚本（都是到/etc/init.d目录中的文件软链接）。

作为Linux系统管理人员，有时候需要根据具体的业务需求自己写init脚本。但是和一般的shell脚本不同，init脚本需要满足一定的格式，最基本的要求是，脚本必须接收至少两个参数：start和stop，分别用于启动和停止服务。系统在启动时所显示的很多的Starting其实是在调用脚本的start参数，如图18-3所示；系统在关机时显示的很多的Stopping则是在调用脚本的stop参数，如图18-4所示。当然，脚本可能由于功能的多样性，还可以接收更多参数。考虑到脚本良好的可读性，建议在写init脚本时，将各种参数的执行体封装成函数的格式。

```

Starting kernel logger:           [ OK ]
Starting irqbalance:             [ OK ]
Starting portmap:                 [ OK ]
Starting NFS statd:              [ OK ]
Starting RPC idmapd:             [ OK ]
Starting system message bus:     [ OK ]
Starting Bluetooth services:     [ OK ]
Mounting other filesystems:      [ OK ]
Starting PC/SC smart card daemon (pcscd): [ OK ]
Starting acpi daemon:            [ OK ]
Starting HAL daemon:            [ OK ]
Starting hidd:                   [ OK ]
Starting autofs: Loading autofs4: [ OK ]
Starting automount:              [ OK ]
                                  [ OK ]
Starting hpiod:                  [ OK ]
Starting hpssd:                  [ OK ]
Starting sshd:                   [ OK ]
Starting cups:                   [ OK ]
Starting xinetd:                 [ OK ]
Starting sendmail:               [ OK ]
Starting sm-client:              [ OK ]
Starting console mouse services: [ OK ]
Starting crond:                  [ OK ]
Starting xfs: _

```

图18-3 开机过程中的服务启动项

```

Broadcast message from root (tty1) (Sat Jun 1 20:57:06 2013):

The system is going down for reboot NOW!
INIT: Sending processes the TERM signal
Shutting down smartd:           [ OK ]
Shutting down Avahi daemon:     [ OK ]
Stopping yum-updatesd:         [ OK ]
Stopping anacron:               [ OK ]
Stopping atd:                   [ OK ]
Stopping cups:                  [ OK ]
Stopping hpiod:                 [ OK ]
Stopping hpssd:                 [ OK ]
Shutting down xfs:              [ OK ]
Shutting down console mouse services: [ OK ]
Stopping sshd:                  [ OK ]
Shutting down sm-client:        [ OK ]
Shutting down sendmail:         [ OK ]
Stopping xinetd:                [ OK ]
Stopping crond:                 [ OK ]
Stopping autofs: Stopping automount: [ OK ]
                                  [ OK ]
Stopping acpi daemon:          [ OK ]
Stopping HAL daemon:           [ OK ]
Stopping system message bus:    [ OK ]
_

```

图18-4 关机过程中的服务停止项

关于init脚本的书写要求，下面通过分析一个简单的系统脚本yum-updatesd来总结。在搞清楚并理解透彻后，自己写init脚本就可以变得很简单了。注意，下面脚本中以##开头的部分是给出的注解。

```
[root@localhost ~]# cat /etc/init.d/yum-updatesd
#!/bin/bash
##
简述一下该脚本的作用，建议有
# yum          Update notification daemon
##
作者的联系方式，建议有
# Author:      Jeremy Katz <katzj@redhat.com>
##
设置chkconfig
，一定要有。其中345
是运行级别为3
、4
、5
时，启动优先级是97
，关闭优先级是03
# chkconfig:   345 97 03
##
更详细的描述，建议要有
# description: This is a daemon which periodically checks fo
#               and can send notifications via mail, dbus or
#
进程名，非必需
# processname: yum-updatesd
# config: /etc/yum/yum-updatesd.conf
# pidfile: /var/run/yum-updatesd.pid
#
##
下面的信息不是必需的，可根据实际情况决定
### BEGIN INIT INFO
# Provides: yum-updatesd
# Required-Start: $syslog $local_fs messagebus
# Required-Stop: $syslog $local_fs messagebus
# Default-Start: 2 3 4 5
# Default-Stop: 0 1 6
# Short-Description: Update notification daemon
```

```

# Description: Daemon which notifies about available updates
#     syslog. Can also be configured to automatically apply
#### END INIT INFO
##
引用库函数
# source function library
. /etc/rc.d/init.d/functions
RETVAL=0
##
定义start
函数的动作，一定要有
start() {
    echo -n "Starting yum-updatesd: "
    daemon +19 'yum-updatesd &'
    RETVAL=$?
    echo
    [ $RETVAL -eq 0 ] && touch /var/lock/subsys/yum-
updatesd
}
##
定义stop
函数的动作，一定要有
stop() {
    echo -n "Stopping yum-updatesd: "
    killproc yum-updatesd
    echo
    [ $RETVAL -eq 0 ] && rm -f /var/lock/subsys/yum-
updatesd
}
##
定义restart
函数，不必需
restart() {
    stop
    start
}
##
脚本主体
case "$1" in
    start)
        start
        ;;
    stop)
        stop
        ;;
    restart|force-reload|reload)
        restart
        ;;

```



```

condrestart|try-restart)
    [ -f /var/lock/subsys/yum-updatesd ] && restart
    ;;
status)
    status yum-updatesd
    RETVAL=$?
    ;;
*)
    echo $"Usage: $0 {start|stop|status|restart|reload|for
reload|condrestart}"
    exit 1
esac
exit $RETVAL

```

按照上面脚本的规范，将前一小节中的防火墙脚本改写成init脚本，如下所示：

```

[root@localhost ~]# cat /etc/init.d/myIptables
#!/bin/bash
#iptables          Control myIptables
#Author:           johnwang.wangjun@gmail.com
#chkconfig:        345 97 03
#description:      This script is for start and stop myIptables
#pidfile:          /var/run/myIptables.pid
#DEFINE VARIABLES
HTTP_PORT=80
SECURE_HTTP_PORT=443
SSH_PORT=22
DNS_PORT=53
ALLOWED_IP=192.168.61.1 #
这是笔者实验环境中主机的地址，读者需要修改成自己的IP
IPTABLES=/sbin/iptables
RM=/bin/rm
PIDFILE=/var/run/myIptables.pid
start() {
    if [ -f $PIDFILE ]; then
        echo "myIptables is running"
        exit 1
    else
        touch $PIDFILE
    fi
    #FLUSH IPTABLES
    $IPTABLES -F
    $IPTABLES -X

```

```

#DEFINE DEFAULT ACTION
$IPTABLES -P INPUT DROP
$IPTABLES -P OUTPUT DROP
#DEFINE INPUT CHAINS
$IPTABLES -A INPUT -p icmp --icmp-type any -j ACCEPT
    $IPTABLES -A INPUT -s localhost -d localhost -
j ACCEPT
    $IPTABLES -A INPUT -m state --
state ESTABLISHED,RELATED -j ACCEPT
    $IPTABLES -A INPUT -p tcp --dport $SSH_PORT -j ACCEPT
#DEFINE OUTPUT CHAINS
$IPTABLES -A OUTPUT -p icmp --icmp any -j ACCEPT
    $IPTABLES -A OUTPUT -s localhost -d localhost -
j ACCEPT
    $IPTABLES -A OUTPUT -m state --
state ESTABLISHED,RELATED -j ACCEPT
    $IPTABLES -A OUTPUT -p tcp -m state --state NEW --
dport $HTTP_PORT -j ACCEPT
    $IPTABLES -A OUTPUT -p tcp --
dport $SECURE_HTTP_PORT -j ACCEPT
    $IPTABLES -A OUTPUT -p udp --dport $DNS_PORT -
j ACCEPT
    $IPTABLES -A OUTPUT -p tcp --dport $SSH_PORT -
j ACCEPT
    echo "Start myIptables OK"
}
stop() {
    if [ ! -f $PIDFILE ]; then
        echo "myIptables is already stopped"
        exit 1
    else
        $RM $PIDFILE
    fi
    #FLUSH IPTABLES
    $IPTABLES -F
    $IPTABLES -X
    #DEFINE DEFAULT ACTION
    $IPTABLES -P INPUT ACCEPT
    $IPTABLES -P OUTPUT ACCEPT
    echo "Stop myIptables OK"
}
case "$1" in
    start)
        start
        ;;
    stop)
        stop
        ;;

```

```
        reload|restart)
            stop
            start
            ;;
*)
    echo "Usage: $0 {start|stop|restart|reload}"
    exit 1
esac
```

注意，该脚本应该放在/etc/init.d目录中，并且要有可执行权限。

编写完成后，使用chkconfig命令添加该脚本成为系统服务。注意：如果系统本身启动了iptables服务，可以将该启动项停用。

```
#
停用系统默认iptables
服务
[root@localhost ~]# chkconfig --level 345 iptables off
#
添加myIptables
为系统服务
[root@localhost ~]# chkconfig --add myIptables
#
使用chkconfig
添加为系统服务后，看到默认345
是开启的，这和脚本中的设置是一致的
[root@localhost ~]# chkconfig --list | grep myIptables
myIptables      0:off  1:off  2:off  3:on   4:on   5:on
#
启动优先级确实是97
[root@localhost ~]# ls /etc/rc.d/rc3.d/ | grep myIptables
S97myIptables
#
关闭优先级确实是3
[root@localhost ~]# ls /etc/rc.d/rc2.d/ | grep myIptables
K03myIptables
```

添加完成后，就可以使用service命令来管理该服务了。

```
[root@localhost ~]# service myIptables start
Start myIptables OK
[root@localhost ~]# service myIptables restart
Stop myIptables OK
Start myIptables OK
[root@localhost ~]# service myIptables stop
Stop myIptables OK
```

18.11 使用脚本操作MySQL数据库

在Shell开发中，有时候需要操作MySQL数据库。比如，定时数据库备份还原任务（导入导出）、数据查询等。另外在很多基于LAMP的开源软件在安装的过程中，也需要通过运行Shell脚本来创建软件运行的数据库环境。本节将演示如何使用Shell脚本操作MySQL数据库，需要读者有一定SQL语法基础。

系统在安装了MySQL客户端后就可以使用mysql命令，并借助-e参数来操作数据库了。示例如下：

```
#
使用mysql
操作数据库
mysql -uUSER -pPASSWORD -e"SQL STATEMENTS"
#
假设本地数据库用户名为root
， 密码为password
， 查看本地所有数据库
[root@localhost ~]# mysql -uroot -ppassword -
e"show databases"
+-----+
| Database |
+-----+
| information_schema |
| mysql |
| test |
+-----+
```

以上操作使用Shell脚本实现时，内容如下：

```
#
使用mysql
操作数据库脚本
[root@localhost ~]# cat mysql01.sh
```

```
#!/bin/bash
HOSTNAME="localhost"
USERNAME="root"
PASSWORD="password"
MYSQL=/usr/bin/mysql
SH_DB="show databases"
$MYSQL -u$USERNAME -p$PASSWORD -e"$SH_DB"
#
脚本运行结果
[root@localhost ~]# bash mysql01.sh
+-----+
| Database          |
+-----+
| information_schema |
| mysql              |
| test               |
+-----+
```

下面列举了常用的数据库操作脚本：

```
#
创建数据库
create_db_sql="create database  ${DBNAME}"
mysql -u${USERNAME} -p${PASSWORD} -e "${create_db_sql}"
#
创建表
create_table_sql="create table  ${TABLE} (name varchar(20), i
mysql      -u${USERNAME}      -p${PASSWORD}      ${DBNAME}      -
e"${create_table_sql}"
#
插入数据
insert_sql="insert into ${TABLENAME} values('john',1)"
mysql      -u${USERNAME}      -p${PASSWORD}      ${DBNAME}      -
e"${insert_sql}"
#
查询
select_sql="select * from ${TABLENAME}"
mysql      -u${USERNAME}      -p${PASSWORD}      ${DBNAME}      -
e"${select_sql}"
#
更新数据
update_sql="update ${TABLENAME} set id=3"
mysql      -u${USERNAME}      -p${PASSWORD}      ${DBNAME}      -
e"${update_sql}"
```

```
#
删除数据
delete_sql="delete from ${TABLENAME}"
mysql -u${USERNAME} -p${PASSWORD} ${DBNAME} -
e"${delete_sql}"
```

使用Here Document执行SQL代码块，命令如下：

```
[root@localhost ~]# cat mysql02.sh
#!/bin/bash
mysql -uroot -ppassword << EOF
CREATE DATABASE DB01;
use DB01;
CREATE TABLE user
(
  userID int(20) not null,
  userName varchar(20) not null,
  userPass varchar(20) not null,
  age int(10) not null,
  primary key(userID)
);
EOF
```

使用管道或重定向符执行SQL代码块，命令如下：

```
mysql -uroot -ppassword < update.sql
cat update.sql | mysql -uroot -ppassword
```

18.12 基于LVM快照的MySQL数据库备份

MySQL数据库是现在网站中流行采用的后台数据库，在Web 2.0时代，网站的核心是数据库——几乎所有网站的内容都存放在数据库中，如果数据库不工作了那就意味着整个网站都无法访问了。

传统备份MySQL数据库的方式是在网站业务低谷时间段（一般是凌晨1~5点），使用mysqldump或mysqlhotcopy进行备份，这种方式往往会造成数据备份不一致的问题。比如说在数据库中存在表A和表B，它们之间存在某种一致性的依赖关系。在备份过程中，当备份工具完成了备份表A的备份之后，在备份表B时，可能A表记录已经发生了变化，这样的备份文件实际上是无法用于数据库恢复的。解决这个问题方法是在整个备份过程中锁定表，直至备份结束，但这会影响网站的可用性，因为在数据库备份过程中由于数据库长时间被锁定而无法写入任何数据。

LVM的快照（snapshot）功能可以很好地解决这个问题。在对一个LV创建snapshot时，仅会复制其中的元数据，而不会有任何真实数据的复制，所以创建过程几乎是实时的。当原LV有写操作时，数据会写到快照中而不是原LV中（写时复制机制，Copy On Write, CoW），从而保证了原LV中数据的一致性。为了确保数据的一致（这里特指MySQL数据），在对其做快照之前也需要对数据库进行锁定操作，做完快照后再解除锁。由于快照的过程极为迅速，所以短时间的数据库锁定并不会对前台应用造成影响。

使用LVM的snapshot功能时特别需要注意的一点是，快照创建的大小必须考虑备份期间数据的变化量，如果数据变化量大于快照的大小则会造成快照损坏。所以建议在对特别重要的

数据进行快照备份时，要让快照的大小必须至少等于原LV的大小。



图18-5 给虚拟机添加磁盘

当然，以上所述都是基于LVM的，使用这个功能的前提是MySQL数据库的数据文件必须存储在LV上，所以本实验需要先准备满足条件需求的环境。如果读者使用的是虚拟机环境，给虚拟机添加一块磁盘并启动（实验时可采用较小的5GB空间）。启动后使用fdisk查看到新添加的磁盘，将其所有空间创建成一个新的VG，并划出其中1GB作为MySQL数据库的新数据目录，如图18-5所示。

请读者参考以下步骤，使用LV替换默认的MySQL数据目录——这是使用快照备份数据库的前提，所以在生产环境规划中如果希望使用LVM的快照功能实现对MySQL的备份，也需要满足MySQL的数据目录是使用的逻辑卷。

```

#
发现新设备/dev/sdb
[root@localhost ~]# fdisk -l
Disk /dev/sda: 5368 MB, 5368709120 bytes
255 heads, 63 sectors/track, 652 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
   Device Boot      Start         End      Blocks   Id  System
/dev/sda1    *           1          13       104391    83   Linux
/dev/sda2           14         652      5132767+    8e   Linux
Disk /dev/sdb: 5368 MB, 5368709120 bytes
255 heads, 63 sectors/track, 652 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
   Device Boot      Start         End      Blocks   Id  System
#
创建PV
[root@localhost ~]# pvcreate MySQL_PV /dev/sdb
Device MySQL_PV not found (or ignored by filtering)
Physical volume "/dev/sdb" successfully created
#
创建VG
[root@localhost ~]# vgcreate MySQL_VG /dev/sdb
Volume group "MySQL_VG" successfully created
#
创建LV
[root@localhost ~]# lvcreate -L 1024M -n MySQL_LV MySQL_VG
Logical volume "MySQL_LV" created
#
格式化挂载
[root@localhost ~]# mkfs.ext3 /dev/MySQL_VG/MySQL_LV
mke2fs 1.39 (29-May-2006)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
131072 inodes, 262144 blocks
13107 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=268435456
8 block groups
32768 blocks per group, 32768 fragments per group
16384 inodes per group
Superblock backups stored on blocks:
    32768, 98304, 163840, 229376
Writing inode tables: done
Creating journal (8192 blocks): done
Writing superblocks and filesystem accounting information: do
This filesystem will be automatically checked every 20 mounts
180 days, whichever comes first.  Use tune2fs -c or -

```

```
i to override.
#
挂载LV
[root@localhost ~]# mount /dev/MySQL_VG/MySQL_LV /mnt
#
关闭MySQL
服务，将/var/lib/mysql/
中的数据全部复制到新创建的LV
中
[root@localhost ~]# service mysqld stop
Stopping mysqld: [ OK ]
[root@localhost ~]# cp -a /var/lib/mysql/* /mnt
#
将/dev/MySQL_VG/MySQL_LV
重新mount
到/var/lib/mysql
，并启动MySQL
[root@localhost ~]# umount /dev/MySQL_VG/MySQL_LV
[root@localhost ~]# mount /dev/MySQL_VG/MySQL_LV /var/lib/mys
[root@localhost ~]# service mysqld start
Starting mysqld: [ OK ]
#
最后不要忘了写/etc/fstab
文件，开机自动挂载该LV
到/var/lib/mysql
```

在给MySQL_LV做快照之前，先使用FLUSH TABLES和FLUSH TABLES WITH READ LOCK强行将所有OS的缓冲数据写入磁盘（类似于操作系统的sync命令），同时将数据库置为全局只读。快照完成之后，再使用UNLOCK TABLES解锁。然后只需要将快照进行挂载，复制其中的数据，在复制完成后删除该快照即可。示例如下：

```
[root@localhost ~]# cat mysql_lvm01.sh
#!/bin/bash
TIMESTAMP=`date +%Y%m%d%H%M%S`
HOSTNAME="localhost"
USERNAME="root"
PASSWORD="root"
MOUNT=/bin/mount
UMOUNT=/bin/umount
LVCREATE=/usr/sbin/lvcreate
```

```
LVREMOVE=/usr/sbin/lvremove
MYSQL=/usr/bin/mysql
TAR=/bin/tar
SNAP_SIZE=1024m
SNAP_MYSQL=SNAP_MySQL_LV
MOUNT_POINT=/mnt
EXEC_MySQL=""
FLUSH TABLES;
FLUSH TABLES WITH READ LOCK;
FLUSH LOGS;
\!      $LVCREATE      --snapshot      --size=$SNAP_SIZE      --
name $SNAP_MYSQL /dev/MySQL_VG/MySQL_LV
UNLOCK TABLES;"
echo "$EXEC_MySQL" | $MYSQL -u$USERNAME -p$PASSWORD -
h$HOSTNAME
$MOUNT /dev/MySQL_VG/$SNAP_MYSQL $MOUNT_POINT
cd /root
$TAR -zcvf ${TIMESTAMP}.tgz $MOUNT_POINT
$UMOUNT $MOUNT_POINT
$LVREMOVE -f /dev/MySQL_VG/$SNAP_MYSQL
```

18.13 页面自动化安装LAMP环境

LAMP环境是Linux+Apache+MySQL+PHP的经典组合，常用来搭建动态网站或者服务器的开源软件。它们本身都是各自独立的程序，但是因为常被放在一起使用，拥有了越来越高的兼容度，因而共同组成了一个强大的Web应用程序平台。随着开源潮流的蓬勃发展，开放源代码的LAMP已经与J2EE和.Net商业软件形成了三足鼎立之势，并且该软件开发的项目在软件方面的投资成本较低，因此受到了整个IT界的关注。从网站的流量上来说，70%以上的访问量由LAMP提供，所以LAMP是最强大的网站解决方案，很多知名网站都采用了该环境，如表18-1所示。

表18-1 部分采用LAMP环境的著名站点

网站	操作系统	Web 服务器	数据库	代码
Yahoo	FreeBSD、Linux	Apache	MySQL	PHP
Facebook	FreeBSD	Apache	MySQL + Memcached	PHP
Wikimedia	Linux	Apache、Lighttpd	MySQL + Memcached	PHP
Flickr	RedHat、Linux	Apache	MySQL + Memcached	PHP、Perl
Sina	FreeBSD、Solaris	Apache、Nginx	MySQL + Memcached	PHP
YouTube	Linux	Apache、Lighttpd	MySQL	Python

由于LAMP环境的流行度非常高，所以LAMP环境的安装也成了很多网站运维人员的基本技能和日常工作内容。笔者曾为国内某著名云计算公司开发了一套可为Linux主机自动安装软件的系统，对外提供相关API调用方法，其中就包含了安装LAMP环境的功能。本节节选了该系统中的部分代码作为示例，带领读者开发一套可完成LAMP环境安装的系统。读者只

需要通过在网页中填写要安装的服务器的IP地址以及该服务器的密码，然后单击按钮，就可以实现LAMP环境的部署。在实际工作时可根据需求再做修改，以实现更多功能。

实验环境：两台预装好Apache、PHP环境的Linux服务器，服务器B（IP：172.16.5.20）模拟前端页面服务器，用于页面化的用户交互；服务器A（IP：192.168.61.130）作为软件安装API服务器，在接到前端服务器安装需求时进行实际的软件安装。在CentOS下只需要一条命令就可以安装完成，如果是没有RHN支持的RedHat系统，则只能采用RPM安装或者按照8.3.3节介绍的方法让RedHat支持yum安装，然后使用yum方式安装。采用yum方式安装的命令如下：

```
[root@localhost ~]# yum -y install httpd php
```

服务器A：提供软件安装API。

```
#
启动httpd
服务
[root@localhost ~]# service httpd start
#
在/var/www/html
目录中创建文件，内容如下所示
#
主程序，用于接收处理前端安装需求
[root@localhost html]# cat install.php
<?php
include_once "crypt.php";
$USER = $_REQUEST['user'];
$CRYPT_PASS = $_REQUEST['pass'];
$PASS = DeCode($CRYPT_PASS, 'D', $key);
$IP = $_REQUEST['ip'];
$OS = $_REQUEST['os'];
$SOFT = $_REQUEST['soft'];
$ID = $_REQUEST['id'];
if ($PASS == "") {
```

```

    print "Error:Password encrypt fail";
    $fp = fopen("log/error.log","a");
    $time = date("D M j G:i:s T Y");
    $fileData = "$time Error:$PASS $IP $OS $SOFT Password";
    fwrite($fp,$fileData);
    fclose($fp);
    exit();
}
if ($USER != "root") {
    print "Error:User Must be root";
    $fp = fopen("log/error.log","a");
    $time = date("D M j G:i:s T Y");
    $fileData = "$time Not root:$PASS $IP $OS $SOFT\n";
    fwrite($fp,$fileData);
    fclose($fp);
    exit();
}
if (($OS == "centos") && ($SOFT == "lamp")) {
    exec("/usr/bin/sudo /var/www/html/nmap_port.sh $IP",$
    if ($status_1 != 0) {
        $fp = fopen("log/error.log","a");
        $time = date("D M j G:i:s T Y");
        $fileData = "$time Error:$PASS $IP $OS $SOFT";
        fwrite($fp,$fileData);
        fclose($fp);
    }
    if ($status_1 == 1) {
        print "Error:Host Unreachable";
        exit(1);
    }
    if ($status_1 == 2) {
        print "Error:Port 80 Running";
        exit(1);
    }
    if ($status_1 == 3) {
        print "Error:Port 3306 Running";
        exit(1);
    }
    exec("/usr/bin/sudo /var/www/html/expect.sh $PASS $IP
    if ($status_2 == 4) {
        $fp = fopen("log/error.log","a");
        $time = date("D M j G:i:s T Y");
        $fileData = "$time Error:$PASS $IP $OS $SOFT
        $status_2, password not correct\n";
        fwrite($fp,$fileData);
        fclose($fp);
        print "Error:IP and password not match";
        exit(1);
    }
}

```

```

    }
    $fp = fopen("log/process.log","a");
    $time = date("D M j G:i:s T Y");
    $fileData = "$time START:$USER $PASS $IP $OS $SOFT $I
    fwrite($fp,$fileData);
    fclose($fp);
    $exec_install="/usr/bin/sudo /var/www/html/install_la
    system("{ $exec_install } > /dev/null &");
    print "Sucess";
}
else
    print "Error:No such soft or OS not support";
?>
#install_lamp.sh
用于安装并启动相关服务
[root@localhost html]# cat install_lamp.sh
#!/bin/bash
IP=$1
SOFT=$2
OS=$3
ssh root@$IP "yum -y install httpd mysql mysql-
server php php-mysql
php-mbstring php-mcrypt php-xml php-xmlrpc php-gd"
ssh root@$IP "/sbin/chkconfig httpd on; /bin/sleep 1 && /sbin
mysqld on; /bin/sleep 1 && /sbin/service httpd start; /bin/sl
/sbin/service mysqld start"
#crypt.php
为加密解密算法，用于对接收到的密码字段进行解密
[root@localhost html]# cat crypt.php
<?php
$key = "!hT3^pDs";
function DeCode($string,$operation,$key='')
{
    $key=md5($key);
    $key_length=strlen($key);
    $string=$operation=='D'?
base64_decode($string):substr(md5($string.$key),0,8).$string;
    $string_length=strlen($string);
    $rndkey=$box=array();
    $result='';
    for($i=0;$i<=255;$i++)
    {
        $rndkey[$i]=ord($key[$i%$key_length]);
        $box[$i]=$i;
    }
    for($j=$i=0;$i<256;$i++)
    {
        $j=($j+$box[$i]+$rndkey[$i])%256;

```



```

        $tmp=$box[$i];
        $box[$i]=$box[$j];
        $box[$j]=$tmp;
    }
    for($a=$j=$i=0;$i<$string_length;$i++)
    {
        $a=($a+1)%256;
        $j=($j+$box[$a])%256;
        $tmp=$box[$a];
        $box[$a]=$box[$j];
        $box[$j]=$tmp;
        $result.=chr(ord($string[$i])^($box[($box[$a]+$bo
    }
    if($operation=='D')
    {
        if(substr($result,0,8)==substr(md5(substr($result
        {
            return substr($result,8);
        }
        else
        {
            return '';
        }
    }
    else
    {
        return str_replace('=',' ',base64_encode($result))
    }
}

```

}
?>

#expect.sh

用于自动复制公钥，请确保已经生成了/root/.ssh/id_rsa.pub

[root@localhost html]# cat expect.sh

#!/bin/bash

PASS=\$1

IP=\$2

auto_ssh_copy_id () {

expect -c "set timeout -1;

spawn /usr/bin/ssh-copy-id -

i /root/.ssh/id_rsa.pub root@\$2;

expect {

(yes/no) {send -- yes\r;exp_continue;}

assword: {send -- \$1\r;exp_continue;}

ssword). {exit 4;}

eof {exit 0;}

}";

}

auto_ssh_copy_id \$PASS \$IP

#nmap_port.sh

用于在安装前测试是否指定服务器已经有相关组件在运行

#

如果是则会导致主程序在运行中途退出，以免发生重新安装造成主机问题

[root@localhost html]# cat nmap_port.sh

#!/bin/bash

IP=\$1

SOFT=\$2

OS=\$3

```
if [ -z ` /usr/bin/nmap -p 22 $IP | grep open | awk '{print $1}'` ]; then
    exit 1
fi
```

```
if [ -z ` /usr/bin/nmap -p 80 $IP | grep open | awk '{print $1}'` ]; then
    exit 2
fi
```

```
if [ -z ` /usr/bin/nmap -p 3306 $IP | grep open | awk '{print $1}'` ]; then
    exit 3
fi
```

```
exit 0
#
```

安装完成后检测应该正常运行的端口，返回主程序特定的退出码

[root@localhost html]# cat last_check.sh

#!/bin/bash

IP=\$1

SOFT=\$2

OS=\$3

COUNT=2

```
if [ -z ` /usr/bin/nmap -p 80 $IP | grep open | awk '{print $1}'` ]; then
    COUNT=$((COUNT - 1))
fi
```

```
if [ -z ` /usr/bin/nmap -p 3306 $IP | grep open | awk '{print $1}'` ]; then
    COUNT=$((COUNT - 1))
fi
```

```
exit $COUNT
#
```

创建日志目录log
文件

[root@localhost html]# mkdir log

[root@localhost html]# touch log/error.log

[root@localhost html]# touch log/process.log

服务器B：提供用户界面。

```
#
启动httpd
服务
[root@localhost ~]# service httpd start
#
在/var/www/html
目录中创建两个文件，内容如下所示
#index.html
， 软件安装的用户页面
[root@localhost html]# cat index.html
<!DOCTYPE          html          PUBLIC          "-
//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
                                <meta    http-equiv="Content-
Type" content="text/html; charset=UTF-8">
                                <title>InstallSoft</title>
</head>
<body>
    <form action="action.php" method="get">
        IpAdd:<input type="text" name="ip" />
        Pass:<input type="text" name="pass"/>
        <select name="os">
            <option value="centos">centos</option>
        </select>
        <select name="soft">
            <option value="lamp">lamp</option>
        </select>
        <input type="submit" value="INSTALL" />
    </form>
</body>
</html>
#action.php
， 用于加密传输过程中的用户密码，以及调用API
的安装方法
#HTTP
包在网络中传输时使用明文传输，所以需要将用户密码进行加密
[root@localhost html] cat action.php
<?php
$key = "!hT3^pDs";
function DeCode($string,$operation,$key='')
{
```

```

        $key=md5($key);
        $key_length=strlen($key);
$string=$operation=='D'?
base64_decode($string):substr(md5($string.$key),0,8).$string;
$string_length=strlen($string);
$rndkey=$box=array();
$result='';
for($i=0;$i<=255;$i++)
{
    $rndkey[$i]=ord($key[$i%$key_length]);
    $box[$i]=$i;
}
for($j=$i=0;$i<256;$i++)
{
    $j=($j+$box[$i]+$rndkey[$i])%256;
    $tmp=$box[$i];
    $box[$i]=$box[$j];
    $box[$j]=$tmp;
}
for($a=$j=$i=0;$i<$string_length;$i++)
{
    $a=($a+1)%256;
    $j=($j+$box[$a])%256;
    $tmp=$box[$a];
    $box[$a]=$box[$j];
    $box[$j]=$tmp;
    $result.=chr(ord($string[$i])^($box[($box[$a]+$bo
}
if($operation=='D')
{
    if(substr($result,0,8)==substr(md5(substr($result
    {
        return substr($result,8);
    }
    else
    {
        return '';
    }
}
else
{
    return str_replace('=',' ',base64_encode($result))
}
}
$password = $_GET["pass"];
$pass = DeCode($password, 'E', $key);
$os = $_GET["os"];
$soft = $_GET["soft"];

```

```
$ip = $_GET["ip"];
$id = rand();
echo
file_get_contents("http://192.168.61.130/install.php?
user=root&pass=
$pass&ip=$ip&os=$os&soft=$soft&id=$id");
?>
```

创建完成后，使用浏览器访问<http://172.16.5.20>，可以看到如图18-6所示页面。

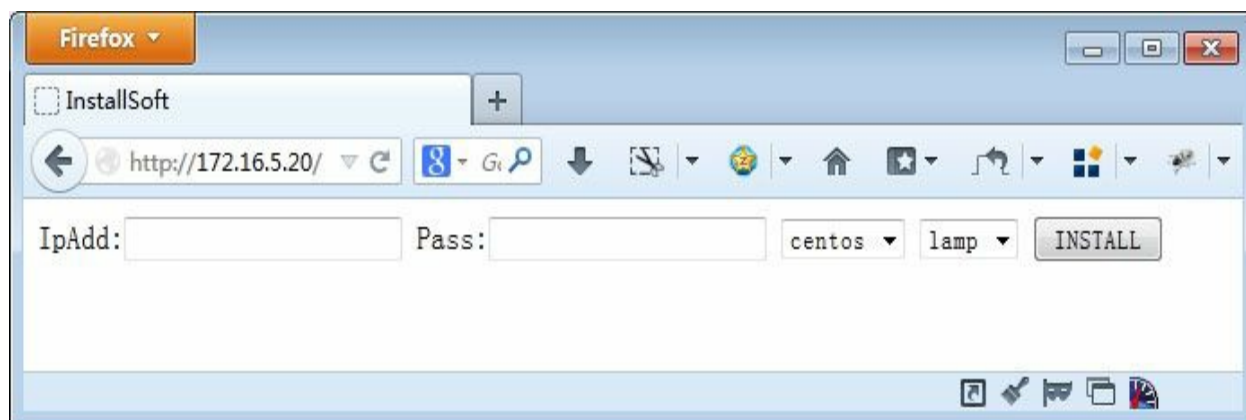


图18-6 软件安装工具前端

只要输入需要安装LAMP环境的主机的IP和root密码，并单击INSTALL按钮，很快该页面就会显示安装结果，如果安装正常则显示Success。常见的几种安装失败场景如下所示：

Error:Host Unreachable	#
主机不可达	
Error:Port 80 Running	#
检测到80	
端口已被占用（已有Web	
服务）	
Error:Port 3306 Running	#
检测到3306	
端口已被占用（已有MySQL	
服务）	
Error:IP and password not match	#
提供的密码不正确，无法登录远程系统	

实测时，该系统在CentOS release 5.5（Final）发行版中能正常运行，但可能会由于系统配置参数等环境差异，而无法保证部署到其他环境中不出现任何问题。如遇运行异常，请查看相关日志（包括但不限于/var/log/message、/var/log/httpd/error_log）。常见如下两个问题，需要对服务器A系统参数进行相关调整。

第一，由于install.php中使用了sudo命令，而Apache服务在运行时使用的是Apache用户，所以默认情况下Apache用户并没有sudo权限，需要通过/etc/sudoers文件给该用户适当的权限，否则PHP脚本会因为权限问题而无法运行。示例如下：

```
apache ALL=(ALL) NOPASSWD:/var/www/html/nmap_port.sh
apache ALL=(ALL) NOPASSWD:/var/www/html/expect.sh
apache ALL=(ALL) NOPASSWD:/var/www/html/install_lamp.sh
#
或者为测试期间方便起见使用下面的设置，但是不建议在生产环境中使用
#apache ALL=(ALL) NOPASSWD:ALL
```

第二，由于Apache用户并不是一个登录用户，所以即便按照上述方法赋予了正确的权限，也可能会因为没有获得tty而无法运行。在这种情况下在Apache的错误日志中会看到如下所示的报错：

```
sudo: sorry, you must have a tty to run sudo
```

解决办法是，注销/etc/sudoers中的Defaults requiretty，表示用户在没有tty的情况下也可以运行命令。命令如下：

```
#Defaults    requiretty
```
